A Query-Based Approach for the Analysis of Aspect-Oriented Systems

by

Eduardo Salomão Barrenechea

A thesis presented to the

University of Waterloo

in fulfillment of the thesis requirement

for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 2007

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

Eduardo S. Barrenechea

I understand that my thesis may be made electronically available to the public.

Eduardo S. Barrenechea

Abstract

In recent years, many aspect-oriented languages and methods have been proposed in the literature to support separation of concerns that can be spread throughout a software system and its components and to facilitate post-development and unpredictable system changes in the code of these systems. These languages and methods provide new abstraction and composition mechanisms to deal with some special concerns, which are called cross-cutting concerns. Cross-cutting concerns, by nature, encode structures that represent changes related to many different system modules, and are often difficult to understand. Also, the provision and support for metrics that can give quantitative estimates related to various software quality features had been a challenge. Because of the complexity and intricate relationships with the base code, techniques for more rigorous analysis are crucially needed to check whether, for instance, some aspects are interfering with other aspects in an undesirable way or not behaving according to the systems requirements and expected behaviour.

In this thesis we advocate that by extending the metrics and analysis capabilities of current approaches, which are often restricted to code-level evaluations, we can (i) define an approach to analyze aspect-oriented systems based on design and architecture-level quality criteria and metrics, (ii) implement tool support for our approach and (iii) provided experimental support based on case studies indicating the usefulness and impact of the approach.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

In recent years, many aspect-oriented languages and methods have been proposed in the literature to support separation of concerns that can be spread throughout a software system and its components and to facilitate post-development and unpredictable system changes in the code of these systems.

These languages and methods provide new abstraction and composition mechanisms to deal with some special concerns, which are called cross-cutting concerns. Such concerns cross-cut the boundaries of other concerns. These cross-cutting concerns include synchronization, presistence, error handling and logging, and deal with issues such as requirements inolving global constraints, system properties and protocols.

Cross-cutting concerns, by their very nature, encode structures that represent changes related to many different system modules, and are often

difficult to understand. In addition, also the provision and support for metrics that can give quantitative estimates related to various software quality features had been a challenge. Because of the complexity and intricate relationships with the base code, techniques for more rigorous analysis are crucially needed to check whether aspects are, for instance, interfering with other aspects in an undesirable way or not behaving according to the systems requirements and expected behaviour.

1.1.1 Aspect-Oriented Programming

Aspect-oriented programming (AOP) is an emerging programming paradigm in computing. Aspect-oriented programming provides a cleaner and simpler way of modularizing concerns that affect different objects and parts of a system. Such concerns are called cross-cutting concerns.

AOP is not a standalone programming paradigm, but is used along with other programming paradigms such as object-oriented programming. Object-oriented programming provides a good degree of separation of concerns, but "it has difficulties localizing concerns which do not fit naturally into a single program module, or even several closely related program modules" [Lee]. AOP is able to provide a solution for this problem by implementing aspects that address these cross-cutting concerns.

Currently there are many different implementations of aspect-oriented languages and frameworks. Some examples for Java are AspectJ [Pro] and HyperJ [J]. There are projects in aspect oriented programming also for C [Aspb], C++ [Aspa], Microsoft's .NET framework [Sha, LOO, Wea] and Smalltalk [Apo] among others.

1.1.2 Software Metrics

Software applications are growing in size and complexity. Software analysis using metrics is an approach to deal with this issue. The use of analysis tools can greatly improve the understanding of how individual modules inside an application work. It provides great insight also into relationships between modules, and help identify possible points of interest such as deadlocks and bottlenecks. Software metrics can provide a basis for comparison between different approaches. Through metrics, engineers are able to evaluate a software application in terms of complexity, quality, adaptability and maintainability.

1.1.3 Object-Oriented Design and Architecture

The software architecture discipline is centered on the idea of reducing complexity through abstraction and separation of concerns. The increase in size of software systems result, among other things, in an increased complexity of the software. As a result, it becomes increasingly harder to understand fully the software system and the relationships between its sub-systems. This affects the implementation, maintenance and evolution of such systems and often results in low quality software being developed.

1.2 Problems

Changes to specific concerns can relate to many disparate and independent system modules, and, for this reason, they are often difficult to understand and analyze. Because of a cross-cutting concern's complexity and intricate

relationships with the base code, techniques for more rigorous analysis are crucially needed to check whether aspects are, for instance, creating new dependencies in the system or even introducing overhead. In addition, there is a need for metrics that can give quantitative estimates related to various software quality features particularly as specific concerns are modified through software system evolution. Providing and supporting these metrics is a challenge.

Most aspects in aspect-oriented software systems represent several concerns that cross-cut multiple modules. For this reason, aspects are in general difficult to understand. In addition, cross-cutting between aspects and components is intricate also because not only can one aspect cross-cut an arbitrary number of components but multiple aspects can cross-cut a single component as well. This many-to-many relationship between components and aspects and the inherent complexity of cross-cutting also makes the problem of measuring and analyzing aspect-oriented systems much harder than the same problem in the case of objects.

In summary, there is a need for extended metric sets and tool support to help developers analyze aspect-oriented systems. Specifically, there is a need for methods and tools to help developers quantitatively evaluate the systems they build in terms of metrics, and qualitatively analyze the systems to check whether they have an undesirable behavior such as an aspect undesirable interference.

Such approaches would be very helpful in many cases. For example, during inspection, a programmer needs to evaluate two or more design options and determine the most simple and efficient one. An approach to analysis,

and its related tools can support him/her in this process and provide a basis for the comparison of these design options.

These problems raise a number of questions, including:

1. Can we provide a query-based approach that can help us to analyze an aspect-oriented system, and its related entities and relationships?

2. How can we develop tool support for the analysis of aspect-oriented systems based on queries, metrics and design criteria?

3. How can we provide experimental results of the analysis of relevant real-world aspect-oriented systems based on our approach?

1.3 Proposed Approach

To alleviate the problems previously mentioned we propose an approach and its related prototype tools that can help developers to analyze aspect-oriented systems. First, AspectJ code is parsed into an internal aspect-oriented representation and stored in an Aspect-Oriented System Knowledge Base (AOS Knowledge Base). This knowledge base contains the systems representation based on the aspect design models we have defined for AspectJ. These models include the essential entities and relationships of AspectJ.

Our approach is supported by a prototype tool called AspectA. A user interface has been defined that allows developers to input queries relating to measurement questions and analysis properties and to view the results. To help support this process, we provide also pre-defined sets of queries, metric definitions, and analysis checks that developers can reuse. In this way, simple

queries or more complex, hybrid questions can be asked involving one or a combination of these features. For example, the developer can pose a query in order to find all aspects that have a pointcut that uses a specific object or can combine this query with the measurement condition that the cohesion of these aspects should exceed.

The approach to aspect analysis consists of the following main steps:

1. Generation of the AOS Knowledge Base. This step consists of parsing the aspect-oriented system in AspectJ to produce the aspect design models that will be stored in the AOS Knowledge Base.

2. Query Definition and Reuse. In this step the features of the system to be analyzed are chosen. Pre-defined analytical queries and metrics can be reused.

3. Query Execution and Evaluation. Using our tool, it is possible to execute the chosen queries and evaluate the results. After query execution and evaluation, there may be a need to go back to Step 2 to define other queries. Query evaluation can provide a valuable input for system re-design based on query results.

1.4 Contributions

In this thesis we advocate that by extending the metrics and analysis capabilities of current approaches, which are often restricted to code-level evaluations, we can *(i)* define an approach to analyze aspect-oriented systems based on design and architecture-level quality criteria and metrics, *(ii)* implement

tool support for our approach and *(iii)* provided experimental support based on case studies indicating usefulness and impact of the approach.

Our approach identified points of interest in the system that could be improved and affect the system at the interface, component and package levels. These changes would help improve modularity, reusability and separation of concerns in the software.

In supporting our thesis statement, we provide the following contributions:

1. The definition of a more comprehensive approach for the analysis of aspect oriented systems based on higher-level analysis and metrics;

2. The definition of a set of criteria, e.g., dead code, redudant execution, dependency constraints, for the analysis;

3. The definition of extended metrics sets, e.g., subsystem dependency, interface level separation of concerns (SoC) and architectural coupling, suitable for the higher-level approach;

4. Implementation of a prototype tool supporting the proposed approach;

5. Development of case studies demonstrating the application and usefulness of the approach.

1.5 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 goes over background information and related work. Chapter 3 describes our approach

and Chapter 4 presents the implementation of our approach. Chapter 5 covers the case studies and experimental results. Finally, Chapter 6 states our conclusions and future work.

Chapter 2

Background and Related Work

2.1 Background

2.1.1 Aspect-Oriented Languages

The world of software engineering is constantly evolving, aiming at facilitating the development of software systems. As software systems became more complex, different better approaches to represent the problem in question were presented. The introduction of higher-level languages enabled programmers to represent their programming goals in higher levels of abstraction from machine language.

Separation of Concerns is defined as the principle of breaking a program into smaller distinct parts without much overlapping in functionality. Separation of concerns is an important concept in software engineering and is

a desired feature in software system modeling and design since it helps in managing the complexity of the system.

Both procedural and object-oriented programming provide ways to apply the separation of concerns principle. They allow for a software system to be decomposed into different modules, each addressing a well-defined concern in the software system. A concern is simply a goal or point of interest in a program. These modules work together to implement the complete solution for the problem being addressed. Object-oriented programming has advantages over procedural programming due to its use of several techniques such as modularity, polymorphism, and encapsulation to decompose and model the problem at hand better, but it still has limitations.

Even though object-oriented programming provides for a good representation of the programming problem, it still has some difficulties in dealing with concerns that are spread throughout different objects. More specifically, if methods relating to different concerns intersect each other [TEO01]. Such concerns are referred to as *cross-cutting concerns*. A more formal definition of cross-cutting concerns is given by Kiczales: *"whenever two properties being programmed must compose differently and yet be coordinated, we say they cross-cut each other"* [GK97]. These types of concerns cannot be easily and cleanly abstracted into objects. Their implementation is either scattered among or tangled with different parts of the system. Examples of such cross-cutting concerns are error handling, synchronization, logging, persistance, security, performance optimizations and memory management.

Cross-cutting concerns go against the separation of concerns principle, since a module is addressing more than one concern in the system. The

tangling and scattering of code results in undesired properties such as an increase in difficulty in maintaining and adapting the modules, increase in module complexity and restriction and oftentimes prevention of module reuse since the module is not addressing a single concern.

Aspect-oriented programming provides a new entity called *aspect*. An aspect is able to isolate cross-cutting concerns completelly and clearly, allowing composition and reuse of modules in an Aspect-Oriented software system. A cross-cutting concern modularized into an aspect has the following properties [Kic03, TEO01]:

1. It is localized;
2. It has a well-defined interface;
3. It allows for separate development.

An aspect-oriented system has two different parts: the *component program* and the *aspect program*.

The *component program* is simply a program using either the procedural or, more commonly, the object-oriented paradigm. It is responsible for identifying and implementing all of the non cross-cutting concerns of the software system. It is responsible also for providing the composition between the component modules.

The *aspect program* is a program using an aspect language. It is responsible for modularizing the cross-cutting concerns of the system. The aspects contain also the composition rules, which is information about composition between aspects themselves and between aspects and components.

Kiczales defines the terms component and aspect as follows [GK97]:

- **Component** if the concern can be cleanly encapsulated in a generalized procedure;
- **Aspect** if the concern cannot be cleanly encapsulated in a generalized procedure.

The actual composition of components and aspects is done by the use of a *weaver*. The weaver takes the component and the aspect programs as input, interprets the composition rules on the aspect program and "weaves" the aspect code into the right places on the right modules. The weaving of the aspect can be done either at compile time or at runtime. The weaver requires that all implementation strategy decisions be provided by the programmer.

Ultimately, the goal of aspect-oriented programming is *"to support the programmer in cleanly separating components and aspects from each other, by providing mechanisms that make it possible to abstract and compose them to produce the overall system"*[GK97].

Aspect-Oriented Languages

In recent years, many aspect-oriented languages and methods have been proposed, with some discussion on what qualifies as an approach to be aspect-oriented [MK03a]. An aspect-oriented language is any programming language that enables a programmer to modularize cross-cutting concerns. This modularization needs to comply with the properties stated in section 2.1.1, mainly localization and a well-defined interface.

In order to modularize cross-cutting concerns, an aspect language needs to have the means to identify particular points on a program and to specify semantics at such points. This ability is implemented through a *join point model*. A join point is any point of interest in the flow of a program which can be, for example, an object instantiation or a method call. Once a join point is identified the specified behaviour can be weaved in the correct place. A join point model can be implemented in many different ways such as class composition, traversal specification [KLO01] and pointcut/advice combinations as used in AspectJ (see section 2.1.1 for more details).

This property is referred to as *quantification* by Filman and Friedman [FF00]. Quantification can be classified as either static quantification or dynamic quantification. Static quantification relates to the conditions that occur on the source code of the program, such as the calling of a method. Dynamic quantification relates to runtime events such as an exception being raised.

Another defining characteristics of aop approaches is obliviouness [FF00]. Since an aspect-oriented system is composed of a component program and an aspect program, the component programmer should be oblivious to the aspect code. This prevents the component programmer from changing the component code to adapt to the cross-cutting concern. This property reiterates the separation of concerns principle.

AspectJ

Even though there are many different implementations of aspect-oriented programming we shall focus on AspectJ [Pro]. AspectJ is an aspect-oriented

language that extends the Java programming language to support modularization of cross-cutting concerns. The base entity where this modularization occurs is called the *aspect*.

AspectJ has two types of cross-cutting mechanisms, one based on a join point model and one based on static cross-cutting. The join point model mechanisms enables the definition of aditional procedures at the join points. This is handled through the use of *pointcuts* and *advices*. Pointcuts are collections of join points combined to pinpoint the location where a cross-cutting concern takes place. Pointcuts are able to expose the data and execution context of a join point. Advices are the procedures that are weaved into the join points. Advices contain the actual code that will address the cross-cutting concerns at a join point. Table 2.1 contains a list of the most common types of pointcuts and advices in AspectJ.

Table 2.1: Common types of pointcuts and advices

Pointcut	**Purpose**
`execution`	Identifies the execution of a method or constructor.
`call`	Identifies the calling of a method or constructor.
`this`	Identifies the object being executed.
`target`	Indetifies operations performed on a given type.
Advice	**Purpose**
`after`	Weaves the code after the join point.
`before`	Weaves the code before the join point.

The static mechanism enables the introduction of new procedures on existing types. These introductions are called *intertype declarations*. AspectJ allows for the introduction of instance variables, constructors and methods into existing objects. It allows the programmer to also change the inheritance hierarchy of an object. When more than one aspect targets the same

pointcut a *declare precedence* statement can be used to order the priority of the aspects.

2.2 Related Work

Research efforts related to our approach fall into the following categories: *(i)* aspect-oriented modeling and languages, *(ii)* metrics and *(iii)* property analysis. In general, our approach provides a more comprehensive analysis of aspect-oriented systems since it addresses different levels of abstraction (e.g., design and architectural packages, design criteria and guidelines), and provides query-based tool support for the analysis of real-world aspect-oriented systems.

In a seminal paper published in 1996, Kiczales et al. have established the conceptual framework for aspect-oriented programming [GK97]. AspectJ is an aspect-oriented extension to Java that was developed to support general-purpose aspect-oriented programming.

2.2.1 Aspect-Oriented Metrics

There are many different proposed approaches to deal with aspect-oriented system and aspect measurement. In [Zha] Zhao defines a metrics suite based on the quantification of the information flow in aspect-oriented systems. He defines a model for aspect-oriented systems that is represented by dependence graphs and bases his metrics on such graphs. Figure 2.1 [Zha] shows an example of such graphs.

The graphs are defined in three levels, namely the module level, the

aspect leve and the system level. The module level refers to individual snipets of code such as advices, intertype declarations and methods. The aspect level refers to an aspect in the system and is composed of many different module level graphs. The system level is the representation of the entire aspect-oriented system. The graphs on the three levels are connected through a series of different types of edges that indicate data and control dependence, parameter passing and containment relations.

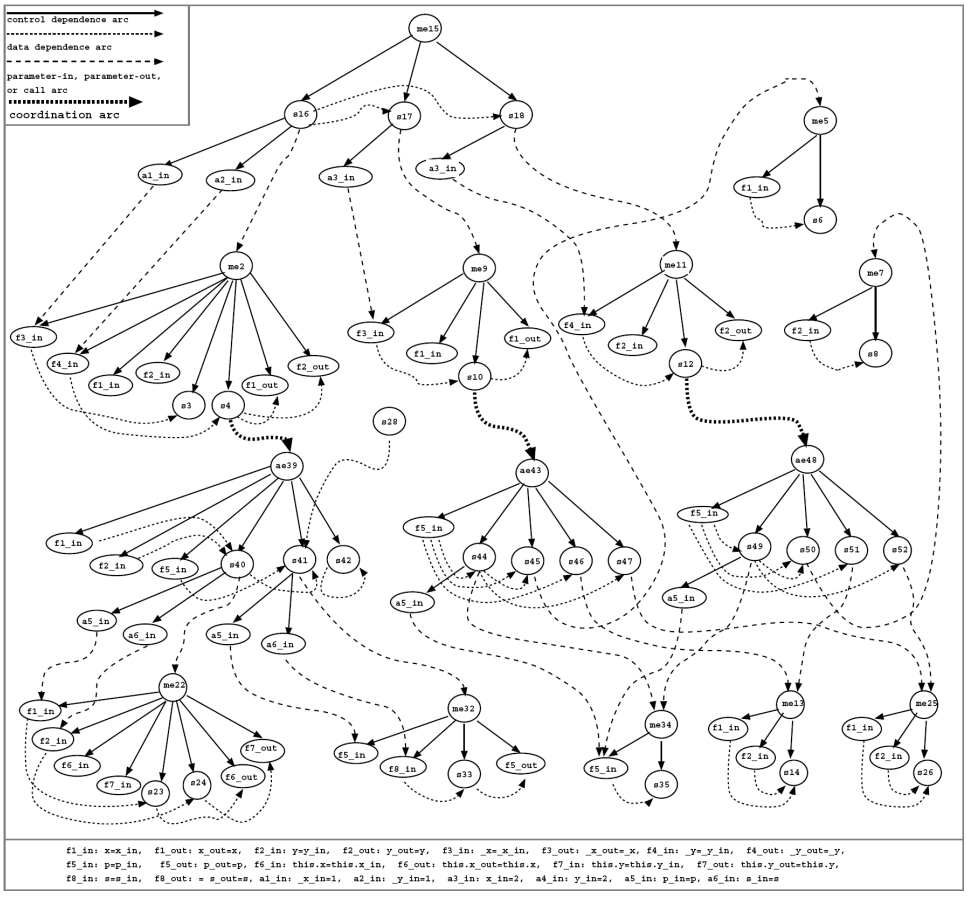

Figure 2.1: System level dependence graph as proposed by Zhao.

His approach, however, does not clearly structures classes and aspects in the aspect-oriented system, but rather follows the information flow throughout the execution of the software. The construction of such graphs is non trivial and time consuming, and may even become infeasible for large aspect-oriented systems. Although his approach enables the calculation of metrics that may assess the complexity of the aspect-oriented system as a whole, it does not provide analysis of the individual aspects and their relationship with other components of the system.

Also Zhao et al. [ZX04] propose a framework for assessing aspect cohesion, based on the analysis of dependencies. This framework is based on the aspect dependence graph, and it analyzes the degree of coherence between aspects attributes and modules (advices, introductions, pointcuts and methods). The approach defines different types of dependencies in an aspect and covers the degree of coherence in three instances: inter-attribute dependencies, module-attribute dependencies and inter-module dependencies.

His approach focuses on the features of the aspect itself, and does not consider the application context in which the aspect is placed. The measurements proposed by this approach are quite complex and the choice of the metric weights is ad hoc [JGB06].

Zhao proposes also a framework for assessing coupling in aspect-oriented systems [Zha04]. The focus of his approach is that the coupling in aspect-oriented systems is due to the degree of interdependence between classes and aspects and the metrics suite is based on counting the dependencies between the classes and aspects in the system. He defines many different dependencies between aspects and classes, but his approach does not take

into consideration dependencies between aspects and aspects nor dependencies between classes and classes.

Another approach to measure aspect cohesion is proposed by Gélinas et al. in [JGB06]. Also their approach is based on dependency analysis. It differs from Zhao's approach in that the dependencies are between module and data and between modules themselves. The cohesion, in this case, is the result of a ratio between connected and non-connected module-data and module-module pairs in an aspect and measures the relatedness of an aspect's module.

In [CSS03] Sant'Anna et al. propose a reusability and maintainability assessment framework based on a metrics suite and a quality model. Their approach defines a metrics suite able to asses different properties of aspect-oriented systems by reusing and extending already defined classical and object-oriented metrics [CK94].

The metrics suite is divided into four categories: separation of concern metrics, coupling metrics, cohesion metrics and size metrics. The separation of concern metrics are related to the identification, encapsulation and manipulation of parts of the software relevant to a given concern. They measure the degree to which a concern is diffused over components, operations and lines of code. The coupling metrics measure the strength of interconnection between components by defining measurements for coupling between components and for depth of the inheritance tree. The cohesion metrics defines a lack of cohesion in a component of the system. The size metrics are based on the measurement of physical properties and length of the software system, such as lines of code and vocabulary size.

Also their proposed assessment framework also contains a quality model. This quality model is based on the analysis of already defined quality models and classical definitions of quality attributes and takes into account the impact of aspect-oriented principles on the software system. The model provides a foundation for the interpretation of the defined metrics and asses the reusability and maintainability of aspect-oriented systems. Although our work takes advantage of some metrics proposed in [CSS03], we extend the basic set of metrics and apply them to aspect-oriented instead of object-oriented code.

Dufour et al. [DGH+03] propose a study of the dynamic behaviour of aspect-oriented software, more specificaly for AspectJ systems, focusing on performance and execution costs. This approach analyzes the overhead introduced by the aspect-oriented language constructs into the system. It is based on the pinciple of assigning tags at compile time that are dynamically propagated through the system. This allows the identification of costly constructs and features with very high accuarcy, correctly identifying system bottlenecks. As a result, guidelines for better performance on AspectJ were proposed. This approach differs from ours in that it focuses on the measurement of dynamic behaviour, and not on static attributes of the source code.

2.2.2 Aspect Analysis and Tool Support

Some tools have been provided for aspect browsing [MK03b, ABUoC, RV], visualization [Han], mining [Han, CZ02, CZJ] and re-factoring [SP04].

QJBrowser

QJBrowser is a source code browser for Java, which is based on a model of source code as a pool of syntactic units represented as logic facts, that supports on-demand re-modularization [RV].

Masuhara and Kiczales have proposed a general framework for aspects [MK03b], which includes the query-based browsing provided by QJBrowser. They focus on a simple mechanism of constructing the hierarchical structures based on queries and lists of variables.

Eclipse AspectJ Development Tools

AspectJ Development Tools (AJDT) is an AsjectJ development suite integrated into the Eclipse IDE, as shown on Figure 2.2. It supports gutter annotations on the code editor to to show where an advice cross-cuts a class and provides other views that portrait the relationships between advices and join points.

Another feature of AJDT is the comparative map, showing how changes in the aspect code, more specifically in pointcuts, affect the cross-cutting of the system, Figure 2.3 shows this tool. AJDT provides also visualization tools that allow cross-cutting concerns to be visualized, as shown on Figure 2.4.

Even though AJDT provides good support for developing AspectJ systems, it still does not allow the system to be queried, and cannot assess detailed properties of the system in question. This follows from the purpose of the tools in AJDT, which is to help developers write software using

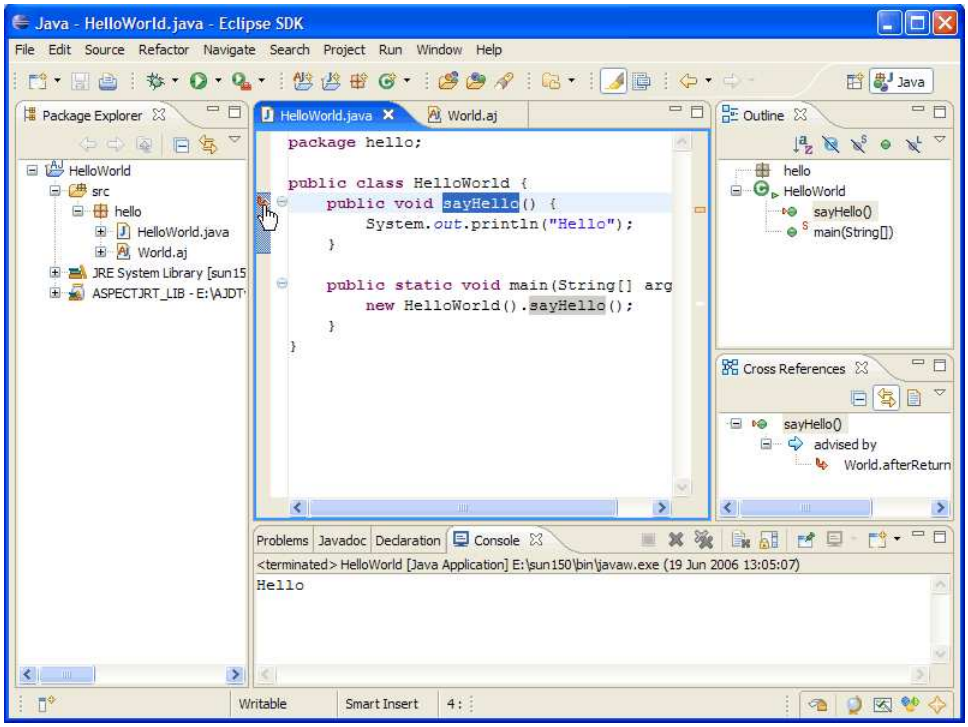

Figure 2.2: The AJDT environment.

AspectJ, rather then focusing on the analysis of AspectJ systems.

Active Aspect

ActiveAspect [Coe] is a tool that produces interactive graphical models of program structures affected by aspects in AspectJ. Figure 2.5 shows the Active Aspect tool. It uses an extension of the UML notation that includes support for aspect-oriented constructs. The models display the direct effects an aspect has on a class, such as inter-type members it declares and the impact of its advices. This model is expandable and able to show additional context such as calls made to introduced members or from advice bodies.

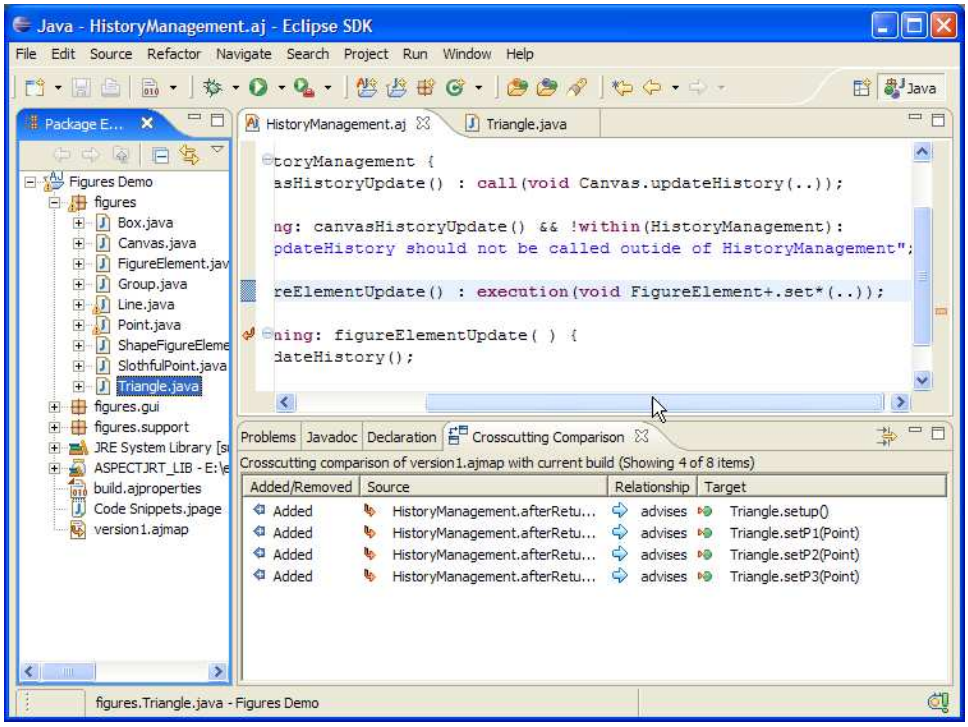

Figure 2.3: The AJDT cross-cutting comparison.

aopmetrics

The focus of the aopmetrics project [aMS] is to provide a metrics tool for both object-oriented and aspect-oriented programming. The project is still under development and aims to provide aspect-oriented extensions to the metrics suite defined by Chidamber and Kemerer [CK94], and by Martin [Mar94].

Aspect Mining Tool

The Aspect Mining Tool (AMT) [Han] implements an analysis framework that supports the identification of concerns as well as system understanding.

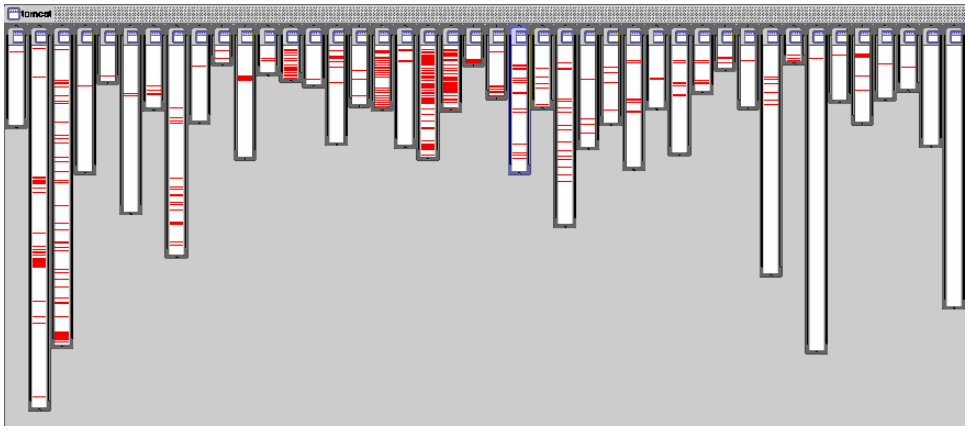

Figure 2.4: The AJDT visualiser.

AMT is composed of two programs, an analyzer and a visualizer, and is able to offers both lexical and type-based analysis techniques.

The analyzer is based on a modified version of the AspectJ compiler. It is extracts all necessary line-oriented program statistics (source code and types used) and other relevant information (package and class hierarchy information) and puts it in a data file.

This data file is used by the visualizer to represent the system in a line-based (i.e. compilation units as collections of lines of code), and supports queries to a system database, which is created by the visualizer, based on the data file. AMT supports the following kinds of queries: substring matching and regular expressions (lexical), and types used (structural). It creates also class and package hierarchies to allow for additional orientation.

Multi-visualizer [Han] is an extension of the visualization functionality of AMT.

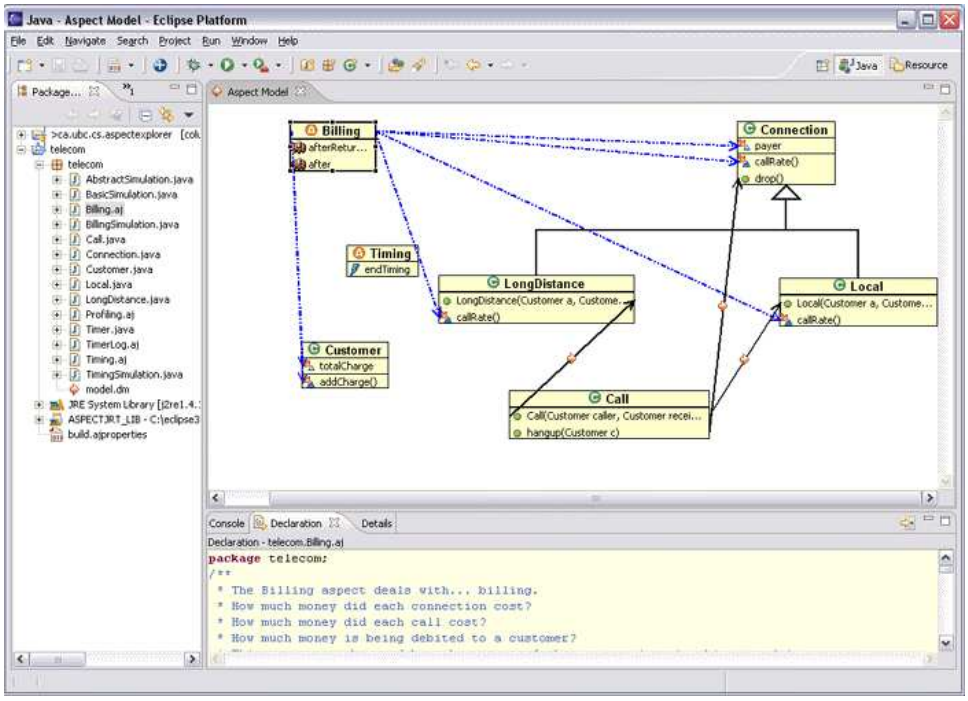

Figure 2.5: The Active Aspect tool.

AspectBrowser

Aspect Browser [ABUoC] helps program visualization by searching for user-defined regular expressions and displaying the results graphically. AspectBrowser includes also features to navigate through search results and manage a potentially large set of regular expressions. Figure 2.6 shows the Aspect Browser.

Prism

Prism offers an aspect mining perspective to the user who can, using this tool, manage mining tasks for the identification of aspects in large Java

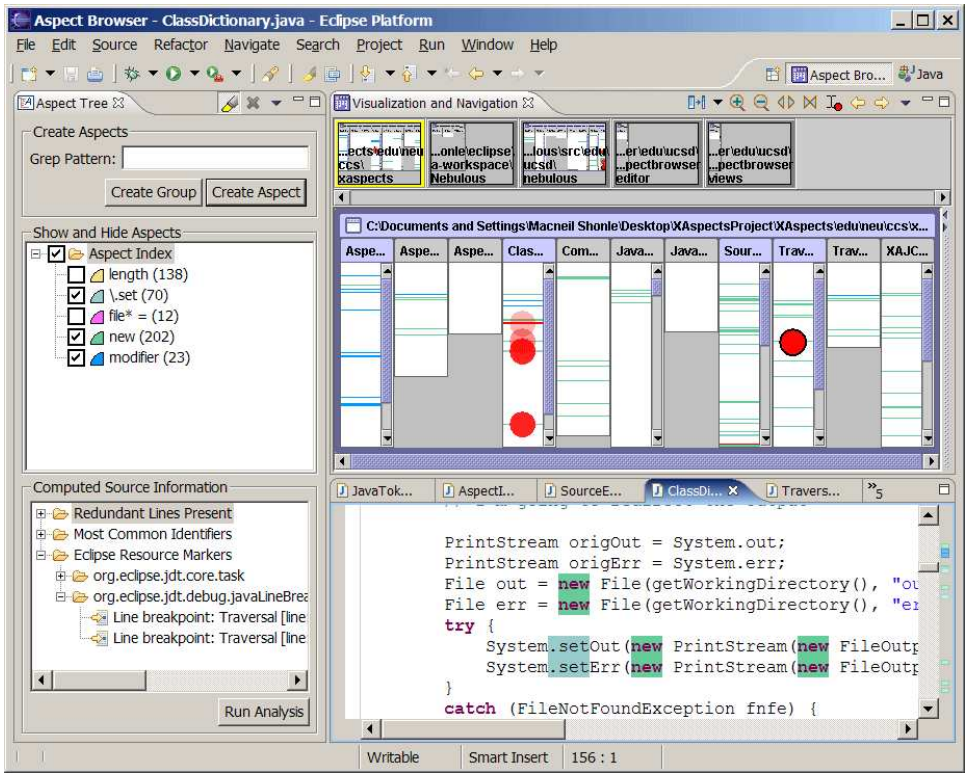

Figure 2.6: The Aspect Browser.

source code bases [CZJ, CZ02].

Ophir

Ophir is a framework for mining which automatically identifies desirable candidates for re-factoring into aspect-oriented programming [SP04].

Chapter 3

A Query-Based Analysis Approach

Our approach consists of defining a model for aspect-oriented systems that represents its structure and properties, and developing a query-based analysis approach and tools to query this model.

Figure 3.1 shows the relationship between the aspect-oriented system, the model and the developer. The aspect-oriented system is compiled into an instance of the model that can be then queried by the developer to extract information about the system.

3.1 Knowledge Base Model

The design goal of the software system model is to provide a simplified yet resourceful representation of the aspect-oriented system that can be queried to help one infer about the structure, relationships and complexity of the

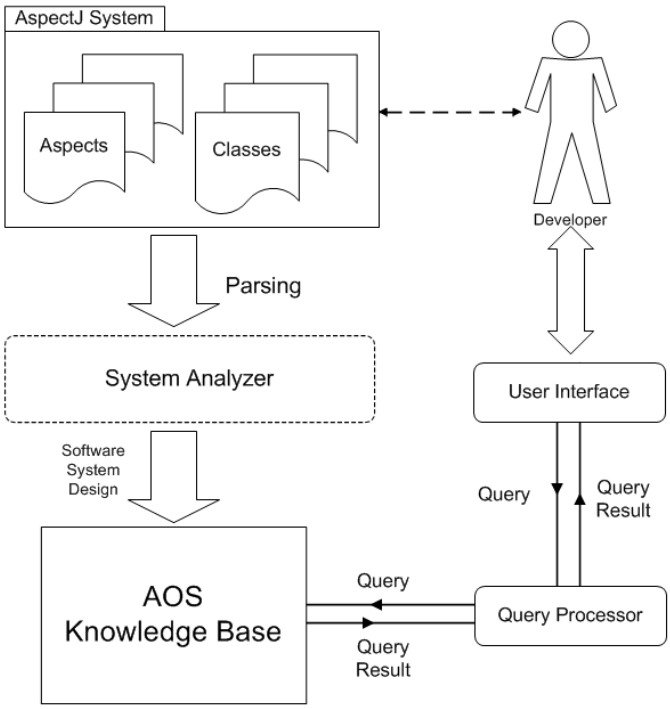

Figure 3.1: Approach overview

entities in this system. To achieve said goal it is necessary that the model faithfully represent each entity of the system. Figure 3.2 shows a hierarchical overview of the software system model.

Our approach organizes the software system as a collection of packages that may contain both classes and aspects. This definition agrees with the notion of an aspect-oriented system being composed of two parts: a component program and an aspect program. Our software system model, therefore, has a collection of package models contains both class and aspect models. Figure 3.3 shows the software system model and the package element using the UML notation.

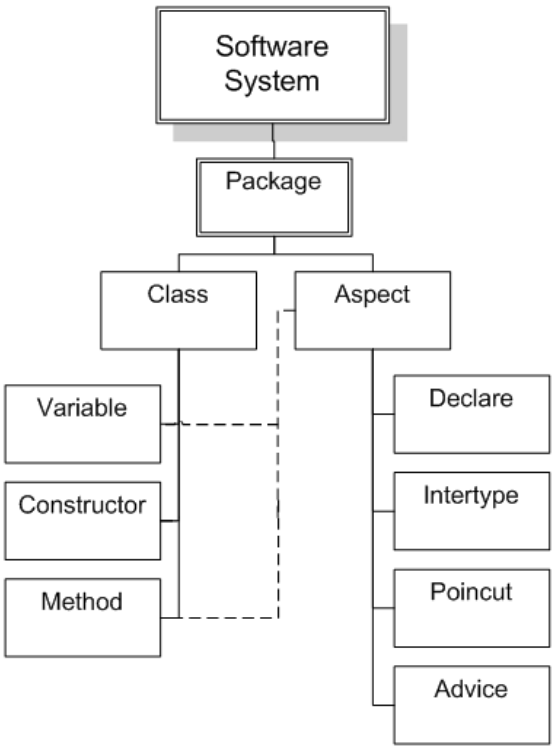

Figure 3.2: Hierarchy of the software system model

The class model[1] represents an entity in the component program. It is able to describe a class, its properties, instance variables and methods. It is able also to infer about relationships between the classes of a system through the analysis of the instance variables and methods of a class (a class being used as an instance variable or as a parameter, for example).

The aspect model represents an entity in the aspect program. It is able to describe the aspect, its properties and cross-cutting behaviour. The aspect model may describe also instance variables and methods. It represents the relationships between the component and aspect programs, through the

[1]We consider the component program to be an object-oriented system.

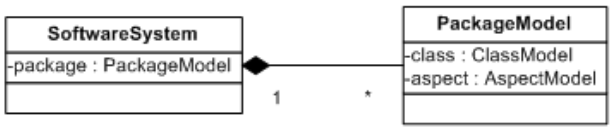

Figure 3.3: Software system model

matching of join points.

The class model does not represent this relationship of matching join points. This design decision is based on the principle of obliviousness of the component program toward the aspect program and it does not hinder the analysis of the software system since relationship information can be reversed.

The model is created by combining features and attributes extracted from the component and aspect source code with information about the weaving of the aspect program into the component program (the relationships). In order to do so we need to examine both the class and aspect definitions first and identify the important constructs that may have an impact on the modeling of the software system.

A class can be defined as a container for abstract characteristics of an entity in the problem domain. The characteristics relate to both attributes and behaviour. An aspect, as defined in Section 2.1.1 and [Kic03, TEO01], is a container for abstract characteristics of cross-cutting concerns in the problem domain. The characteristics relate not only to attributes and behaviour, but also to localization of such cross-cutting concerns.

Although there are two distinct major entities (classes and aspects), they both share some of the same characteristics, and may co-exist inside

the same packages. An aspect can be considered an extension of the class entity that defines a join-point model.

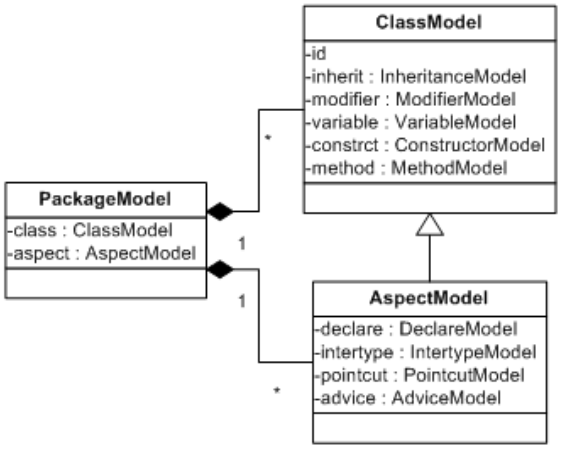

Figure 3.4: Package model

In the following sections we describe the class model and the aspect model in detail and state the information extracted to create the entire software system model.

3.1.1 Class Model

The class model is used to represent classes or interfaces in the system. The class model is composed of a modifier model, an inheritance model, a variable model, a constructor model and a method model. Figure 3.5 shows the class model.

The modifier model is responsible for representing the class properties, such as scope visibility or being an abstract class. The inheritance model, shown on figure 3.6, is differentiated between two categories: the implement model and the extend model. The only difference between the two is that

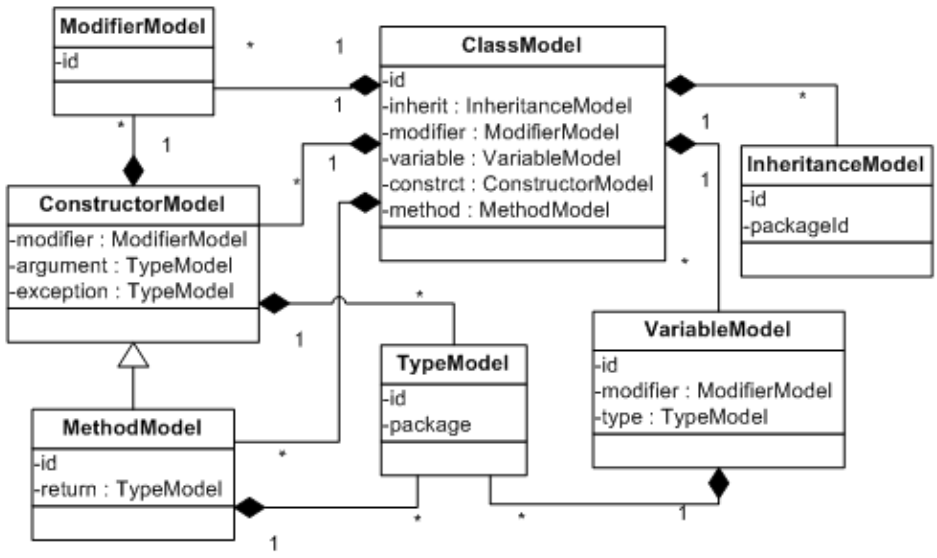

Figure 3.5: Class model

the first is used to indicate the implementation of an interface, while the later indicates the extension of a class.

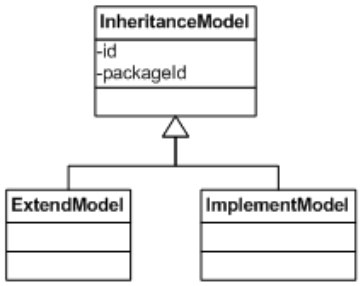

Figure 3.6: Inheritance model

The other components of a class model are the variable, constructor and method elements, and deal specifically with the representation of those entities in the model.

Variable Model

The variable model is used to represent instance variables of a class. It is shown in figure 3.7. It uses modifier elements to represent the scope visibility and extra properties. The type model is used to fully identify the declared object type of the variable through a class name and package combination.

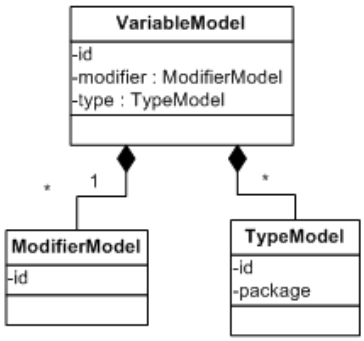

Figure 3.7: Software system model

Constructor and Method Model

The constructor model is used to represent the class constructor. A constructor may have only scope visibility modifiers, which are represented through the modifier model. The constructor arguments as well as any throwable exception are modeled through the use of a type model. In our model, the method is an extension of the constructor, with the inclusion of a modifier and a return type modeled by a type model. Even though there's a conceptual difference between a constructor and a method, from the point of view of the model and its purpose this definition is not incorrect.

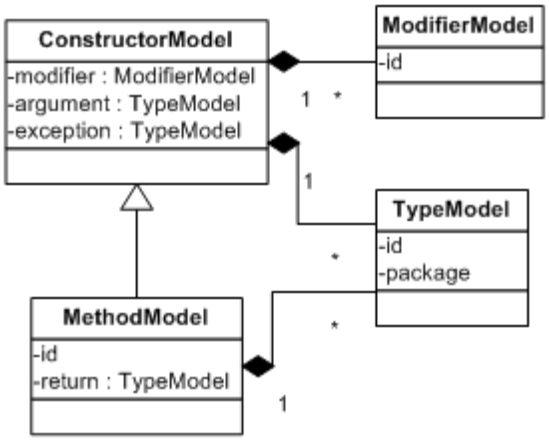

Figure 3.8: Constructor and method model

3.1.2 Aspect Model

The aspect model is based on the class model, since we can consider an aspect as an extension of the class entity (see Section 3.1). Figure 3.9 shows the aspect model. The aspect model represents a modularized cross-cutting concern. It consists of the all the models declared in the class model (except for the constructor model), along with a declare model, an intertype model, a pointcut model and an advice model.

Static Cross-cutting

The declare model is used to represent the declare statements of the aspect-oriented language. The declare model supports three types of declare statement. The first one is the declare precedence, and uses the declaration element to identify the order in which the aspects should process a given join point. The second one is the declare soft, which uses the match model

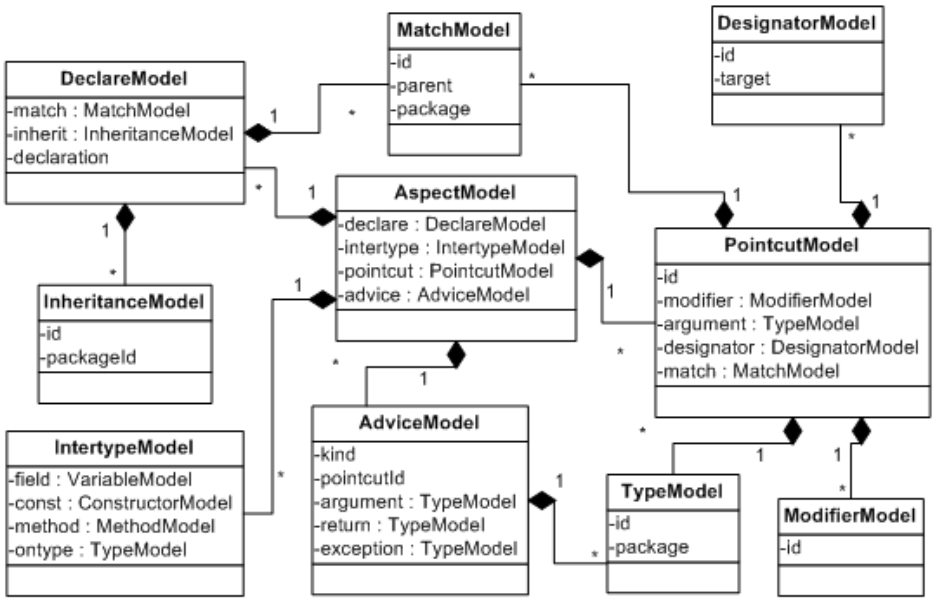

Figure 3.9: Aspect model

to specify join points where it will soften an exception, if it is thrown. The third is the declare parents, which uses the inheritance model to indicate that a certain class either extends another or implements an interface. The declare parents is closer in essence to intertype declarations.

The intertype model corresponds to the declaration of instance variables, constructors and methods existing classes defined in the component program. The intertype model specifies either a variable, constructor or method to be included in a class through their respective models and uses a type model to indicate its target class.

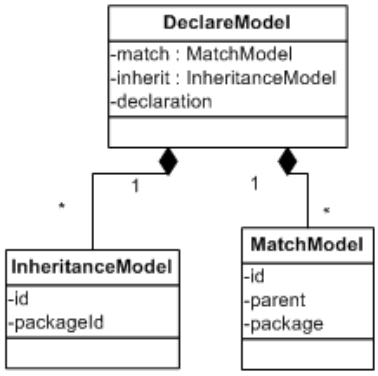

Figure 3.10: Declare model

Pointcuts and Advices

The pointcut model contains a modifier model, a type model, a designator model and a match model. Figure 3.11 shows the pointcut model. The modifier model represents the pointcut properties and scope visibility, while the type model represents the pointcut arguments.

The designator represents the primitive pointcuts that compose the join point. It is composed of an identifier, which indicates the pointcut type, and a target which holds the pattern of the join point. The match model represents a match for the target pattern in the system. It is composed of the target, the parent element and the package identifiers. The target identifier refers to the method or instance variable name, for example. The parent element indicates the class where the target is declared and the package identifies the location of the class in the system.

The advice model specifies a kind and a pointcut identifier. It contains a type model that represents the arguments the advice may receive, the return type, and any exception that may be raised. The advice identifies when the

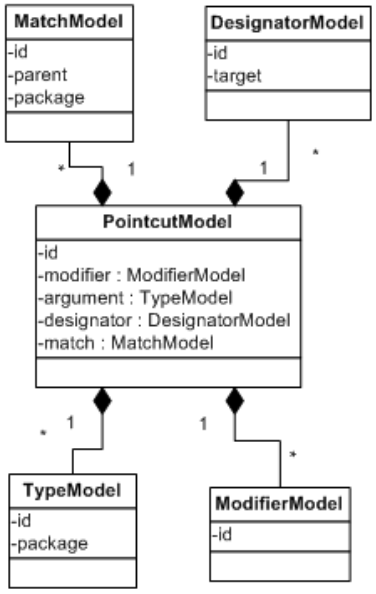

Figure 3.11: Pointcut model

cross-cutting concern should be applied in the target, either before, around or after the target.

3.2 Query-Based Analysis

In this section we describe queries related to analyzing the software system in its different levels of granularity. We provide sample queries to help understand the system from design and architectural level, from an object relations level and from a cross-cutting level.

3.2.1 System, Aspect and Class Analysis

System Model Queries

The architecture of a software system closely relates to the concept of separation of concerns, and software systems are usually divided into packages that address a specific concern. Take, for example, an internet browser application. It is divided into packages that deal with different concerns such as networking, user interface and html rendering. Even though those concerns can be considered as higher levels concerns, they are still present in the architecture of the software system. The packages, in turn, perform their own internal separation of concerns, and address different faces of the major concern. The networking package in our internet browser example could be further decomposed into sub-packages, for example one that implements the required protocols and another that controls the socket connections.

Our approach defines a software system model. This model is able to capture relationships between packages, classes and aspects and represent the structure of an aspect-oriented system. Our query based approach allows us to query this model and extract information about those relationships and about the structure of the system and its components. The result is a better

understanding of the system as a whole and insight into the complexity of the software system.

Query	Query Description
S1	Find all packages in a system.
S2	Find all sub-packages of a package.
S3	Find all classes and aspects of a package.
S4	Find all dependencies between two packages.
S5	Find all packages affected by an aspect.

Table 3.1: Software system related queries.

Our software system model supports queries such as the ones listed in table 3.1. The introduction of aspect-oriented constructs in a system may result in the creation of dependencies between packages that did not exist previously. This new dependency may interfere with the overall architecture and separation of concerns defined by the system. The use of queries and may help identify how a cross-cutting concern behaves in the system and may even provide guidelines to modularize this cross-cutting concern better, in conformance with the architecture of the system.

Aspect and Class Model Queries

The aspect and class model are used to represent the attributes, characteristics and relationship of aspects and classes in the software. Classes represent low level concerns in a system, while aspects represent cross-cutting concerns in a system. Our query-based approach enables one to query both classes and aspects according to their structure and relationships and attain a better understanding of the concerns they represent and their interactions in order to address the higher level concerns.

Query	Query Description
C1	Find all variables/methods in a class.
C2	Find all arguments from a class constructor.
C3	Find all methods that have a given class as argument.
C4	Find all classes that use a given class as method argument.
C5	Find all superclass/interfaces a class inherits from.
C6	Find all classes that implement a given interface.
C7	Find all exceptions thrown from a class.
C8	Find all the classes that throw a given exception.
C9	Find the return type of all methods in a class.
C10	Find the classes that return a given class.

Table 3.2: Class related queries.

The class model is used to support queries such as the ones listed in Table 3.2. By analyzing the components of a class, it is possible to identify the relationship between objects. This provides the means to understand the system without taking any cross-cutting concerns into consideration, in other words, one is able to analyze and understand the system in a way that is oblivious to the cross-cutting concerns that are present in the system.

The aspect model is used to support queries such as the ones listed in Table 3.3. When analyzing the aspects, we are in fact analyzing the cross-cutting concerns in the system. Those cross-cutting concerns may occur with different levels of granularity. For example, a synchronization aspect cross-cutting different thread-enabled classes versus a logging aspect that cross-cuts entire packages or even the whole system.

Our approach allows us to extract information about the aspect and their join points in a system. Using this information it is possible to understand the roles of the classes in relation to the cross-cutting concern and how they are affected by this cross-cutting concern. It supports pointcuts, advices

Query	Query Description
A1	Find all pointcuts/advices in an aspect.
A2	Which advices use a given pointcut?
A3	Find all aspects that affect a given class or method.
A4	Find all classes used by an aspect.
A5	Find all aspects that have pointcuts using a given class.
A6	Find all methods declared in an aspect.
A7	Find all pointcuts that use a primitive element.
A8	Find all pointcuts that are used in an after/around/before advice.
A9	Find all the join point matches per pointcut.
A10	Find all classes that are returned in around advices.
A11	Find all advices that throw an exception.
A12	Find all advices that use the same pointcut.
A13	Find all introduced fields.
A14	Find all introduced methods.
A15	Find all introduced fields in an aspect.
A16	Find all introduced methods in an aspect.
A17	Find all aspects that implement an interface.
A18	Find all interfaces that an aspect implements.
A19	Find all abstract aspects.
A20	Find all aspects that extend a given aspect.

Table 3.3: Aspect related queries.

and static cross-cutting constructs.

3.2.2 System Metrics

Our approach can be used also to extract some metrics from the system through the use of queries. In [CSS03] and [Mar94], both Sant'Anna et al. and Martin define some metrics that can be retrieved from our model:

Size Metrics

Vocabulary Size: the number of system components (classes and aspects).

Number of Attributes(NoA): the number of internal attributes of each component.

Weighted Operations per Component(WOC): measures the complexity of a component by counting the number of arguments of the operation. This assumes that an operation with more arguments is likely to be more complex than one with fewer arguments.

We can expand further in some of those concepts:

Cross-cutting Vocabulary Size(CVS): the number of system components that deal specifically with cross-cutting concerns, ie. the number of aspects in the system. This is useful when assessing the extent of the influence of cross-cutting concerns in the system.

Number of Cross-cutting Attributes(NoCA): the number of internal vocabulary directly connected with identifying cross-cutting concerns. In other words, the number of pointcuts.

Weighted Primitive Pointcuts(WPP): the number of primitive pointcuts that comprise a pointcut. This metric is based on the Weighted Methods per Class metrics [CK94]. It follows from the principle that the more primitive pointcuts are combined to create a pointcut, then the more complex it becomes.

Coupling Metrics

Coupling Between Components(CBC): count of the number of times a class is used as a variable or argument.

We can extend further the concept to accomodate also cross-cutting:

Cross-cutting Coupling(CCC): the number of aspects that target the same join point in the system. A join point is related to many aspects, then a change in the join point may result in a change also in the aspects.

Package Dependency Metrics

Abstractness (ABS): The ratio of the number of abstract modules to the total number of modules in the package. The metric has the range of [0,1], with 0 indicating a completely concrete package and 1 indicating a completely abstract package.

Afferent Couplings(CA): The number of classes and aspects outside this category that depend upon classes within this category. This includes use as an instance variable, method or advice argument and return types.

Efferent Couplings(CE): The number of classes and aspects inside this category that depend upon classes outside these categories. This includes use as instance variable, method or advice argument and return types.

Instability(I): This metric has the range [0,1]. $I = 0$ indicates a maximally stable category. $I = 1$ indicates a maximally unstable category. Instability is computed as follows: $I = \frac{CE}{CE+CA}$

The measurement techniques include the metrics described in Table 3.4

Metric	Metric Name
M1	Number of pointcuts.
M2	Number of aspects.
M3	Cross-cutting vocabulary size.
M4	Number of pointcut arguments - by aspect.
M5	Number of pointcut arguments - by pointcut.
M6	Number of pointcut arguments - by object.
M7	Number introductions - both attributes and methods.
M8	Number introductions - attributes.
M9	Number introductions - methods.
M10	Number of internal methods and attributes - both.
M11	Number of internal attributes.
M12	Number of internal methods.
M13	Weighted operations per component.

Table 3.4: Metrics related queries.

3.2.3 Design Guidelines

In [DGH+03] Dufour et al. defined some guidelines for the usage of AspectJ, in particular. According to their research, some AspectJ constructs tend to add significant overhead to the execution of the code. Four points are of particular interest: loose pointcuts, unwarranted use of `around` advices, use of `cflow` pointcuts and use of `pertarget` pointcuts. The first guideline, loose pointcuts, deals with pointcuts that have many join points. The second one states that the generic form of `around` advices can introduce significant overhead due to the boxing and unboxing of arguments passed to the advice. The use of `after returning` advices is preferred over the use of `around` advices. The third and fourth deals with the use of `cflow` and `pertarget` primitive pointcuts. `cflow` pointcuts also introduce significant overhead and the use of a `withincode` pointcut is preferred. `pertarget` pointcuts are used to control the creation of aspects, and this introduces overhead also in the

compiled system.

In many cases, the aspects defined in AspectJ interfere with other aspects in an undesirable way. This interference may lead to problems that are not detected by the AspectJ compiler such as the introduction of 'dead' code or multiple execution of the same advice. These problems are indicated in aspect-oriented and AspectJ mailing lists and newsgroups, but their non-automatic detection is very challenging.

The first problem we describe is that an `around` advice can block a `before` or `after` advice related to the same join point. In this case, the `around` advice can cause the `before` or `after` advice to become 'dead' code. The second problem[2] is that if there are concrete pointcuts inside an abstract aspect, the advice related to each of these pointcuts is executed $n+1$ times, where n is the number of concrete instances of the abstract aspect.

Properties to check each one of these interference types and guidelines are described in table 3.5.

P1	Find pointcuts with more than x join point matches.
P2	Find all around advices.
P3	Find all pointcuts using a cflow primitive.
P4	Find all pointcuts using a pertarget primitive.
P5	Find if there exists an around advice that can block a before or after advice related to the same method.
P6	Find if there exists any concrete pointcut inside any abstract aspect.

Table 3.5: Design guidelines.

[2]Issue present in early versions of AspectJ

Chapter 4

Implementation

The implementation of our approach deals with the parsing of source code and gathering of relationships to implement the software system model in the form of a knowledge base. Since there are different implementations of aspect-oriented programming, each with its own syntax and constructs, we limited our implementation to the modeling of aspect-oriented systems developed in Java and AspectJ. Some of the reasons for this are:

- AspectJ is one of the most advanced implementations of aspect-oriented programming;
- AspectJ has a large user base compared to other aspect-oriented languages;
- AspectJ is open source, with the source code readily available;
- AspectJ has good documentation on language constructs and syntax;

The second part is to query this knowledge base and extract information about the system. Many different approaches are suitable for our needs in terms of storing and retrieving information, with database systems being the strongest contenders. Full fledged database systems have also their drawbacks, and portability and flexibility being the most important. A non-traditional database such as XML is far more attractive for our purposes.

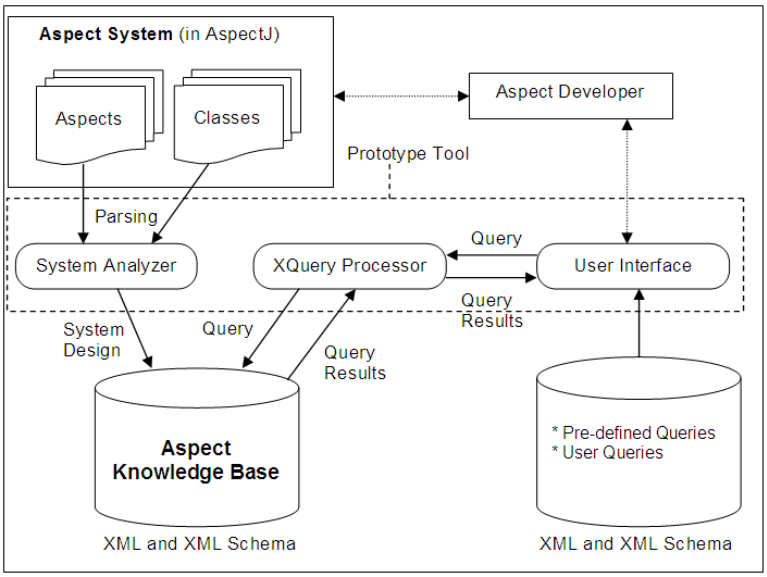

Figure 4.1: Implementation of the approach.

XML and its related technologies and tools offer a powerful and standard way to deal with semi-structured data. Consequently, we chose XML as the structure to represent our AOS Knowledge Base, and XML Schema language [W3C] to define and implement the data schema (i.e. the valid class of data that can be stored). The XML Schemas are extensible to incorporate future additions since they introduce inheritance of complex types.

4.1 Knowledge Base XML Schema

The XML Schema language [W3C] to define the xml schema for our knowledge base is created using the software system model defined in section 3.1. The full schema can be found in appendix B.

4.1.1 Software System and Package Elements

The schema follows the definition of a software system as a collection of packages, and packages being containers of classes and aspects. Table 4.1 shows the software system and package elements.

```
<xs:complexType name="SoftwareSystemModel">
 <xs:sequence>
  <xs:element name="package"
              type="PackageModel"
              minOccurs="1"
              maxOccurs="unbounded" />
 </xs:sequence>
</xs:complexType>

<xs:complexType name="PackageModel">
 <xs:sequence>
  <xs:element name="class" type="ClassModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="aspect" type="AspectModel"
              minOccurs="0" maxOccurs="unbounded" />
 </xs:sequence>
 <xs:attribute name="id" type="xs:string"
               use="required" />
</xs:complexType>
```

Table 4.1: The software system and package elements.

Each package has its own identifier, the package id, which uniquely iden-

tifies the package within the software system. All of the classes and aspects contained in a package have the property of belonging to that given package. This creates the notion of a subsystem. Even though packages do not contain other packages, this notion is embedded in the package id, since Java and AspectJ treat packages as a series of directories in the file system. An inner package is a package that contains the path of the outter package in its identifier. For example, package `org.eclipse.ajdt` belongs to the package `org.eclipse`.

4.1.2 Class Element

The class element is shown on table 4.2. It has an identifier, which uniquely identifies the class within its containing package. A class may have more than one modifier, since modifiers are not only restricted to scope modifiers, but also to other properties such as being an abstract class or a final class. The list of possible class modifiers is: `abstract, final, private`[1], `protected`[1], `public` and `strictfp`. A class element may implement many interfaces, but is allowed to have only one superclass. It has also collections of instance variables, constructors and methods.

Variable Element

Table 4.3 shows the variable and type elements. A variable needs a unique identifier inside its declaring class. It requires also modifiers that, like the class element, are not limited to scope visibility. The list of possible variable modifiers is: `final, private, protected, public, static, transient`

[1]Possible only if it is an inner class

```
<xs:complexType name="ClassModel">
 <xs:sequence>
  <xs:element name="modifier" type="ModifierModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="extends" type="ExtendsModel"
              minOccurs="0" maxOccurs="1" />
  <xs:element name="implements"
              type="ImplementsModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="variable" type="VariableModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="constructor"
              type="ConstructorModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="method"
              type="MethodModel"
              minOccurs="0" maxOccurs="unbounded" />
 </xs:sequence>
 <xs:attribute name="id" type="xs:string"
               use="required" />
</xs:complexType>
```

Table 4.2: The class element.

and `volatile`. A variable has also a type, i.e. is an instance of a given class defined in a given package. This is represented through the type element, which states the class that the variable is instantiating and the package such class belongs.

Constructor and Method Elements

The constructor element does not require an identifier, since it is automatically invoked through the use of the `new` keyword. It may have only the scope visibility modifiers `private, protected` and `public`. It contains also a col-

```
<xs:complexType name="VariableModel">
 <xs:sequence>
  <xs:element name="modifier" type="ModifierModel"
              maxOccurs="unbounded" />
  <xs:element name="type" type="TypeModel"
              maxOccurs="1" />
 </xs:sequence>
 <xs:attribute name="id" type="xs:string"
               use="required" />
</xs:complexType>

<xs:complexType name="TypeModel">
 <xs:attribute name="id" type="xs:string"
               use="required" />
 <xs:attribute name="package" type="xs:string"
               use="required" />
</xs:complexType>
```

Table 4.3: The variable model.

lection of arguments and a collection of exceptions. The method element is an extension of the constructor element, and includes (1) an identifier that uniquely identifies the method within the declaring class and (2) a return element. The return element states the type of the object being returned, as well as the declaring package for that object. In addition to the scope visibility, a method can have the following modifiers: `abstract, final, native, static, synchronized` and `strictfp`.

The constructor and method elements are shown on table 4.4.

4.1.3 Aspect Element

The aspect element is implemented as an extension of the class element with the additional support for dynamic and static cross-cutting. Since it

```
<xs:complexType name="ConstructorModel">
 <xs:sequence>
  <xs:element name="modifier" type="ModifierModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="argument" type="ArgumentModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="exception" type="ExceptionModel"
              minOccurs="0" maxOccurs="unbounded" />
 </xs:sequence>
</xs:complexType>

<xs:complexType name="MethodModel">
 <xs:complexContent>
  <xs:extension base="ConstructorModel">
   <xs:sequence>
    <xs:element name="return" type="TypeModel"
                maxOccurs="1" />
   </xs:sequence>
   <xs:attribute name="id" type="xs:string"
                 use="required" />
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
```

Table 4.4: The constructor and method elements.

inherits all of the elements and attributes from the class element, it has also a unique identifier within its containing package, supports inheritance and collections of instance variables and methods. Aspects do not have constructors, though, but since this element is optional on the class element it will not affect aspect element in any way. An aspect element may have the following modifiers: `abstract, final, private, privileged, protected, public` and `static`[2]. An aspect element may contain a collec-

[2]All inner aspects must be declared static

tion of declarations, intertype definitions, pointcuts and advices. Table 4.5 shows the aspect element.

```
<xs:complexType name="AspectModel">
 <xs:complexContent>
  <xs:extension base="ClassModel">
   <xs:sequence>
    <xs:element name="declare" type="DeclareModel"
                minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="intertypefield"
                type="IntertypeFieldModel"
                minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="intertypemethod"
                type="IntertypeMethodModel"
                minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="intertypeconstructor"
                type="IntertypeConstructorModel"
                minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="pointcut" type="PointcutModel"
                minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="advice" type="AdviceModel"
                minOccurs="0" maxOccurs="unbounded" />
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
```

Table 4.5: The aspect element.

Declare Element

The declare element defines a declare statement in an aspect. Its kind can be of the types `soft` and `parent`. The extends and implements elements are used in conjunction with only the `parent` statement, while the match element is used in conjunction with the `soft` statement. The match element

is discussed in more detail in the *Pointcut Element* subsection. In this instance, the match element identifies a pointcut where an exception could be thrown (the execution or call of a method or constructor, for example). Table 4.6 shows the declare and match elements.

```
<xs:complexType name="DeclareModel">
 <xs:sequence>
  <xs:element name="match" type="MatchModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="extends" type="ExtendsModel"
              minOccurs="0" maxOccurs="1" />
  <xs:element name="implements" type="ImplementsModel"
              minOccurs="0" maxOccurs="unbounded" />
 </xs:sequence>
 <xs:attribute name="kind" type="xs:string"
               use="required" />
</xs:complexType>
```

Table 4.6: The declare and match elements.

Intertype Elements

An intertype declaration is the addition of a method, constructor or instance variable in a class. Each of those is already defined in the class element, and their intertype counterparts extend those elements to add class identification capabilities. This is achieved through the ontype element, which is used to identify a class and package where the intertype should be inserted. Table 4.7 shows the ontype element.

Table 4.8 shows the intertype field and intertype method elements.

```
<xs:complexType name="OnTypeModel">
 <xs:attribute name="id" type="xs:string"
               use="required" />
 <xs:attribute name="package" type="xs:string"
               use="required" />
</xs:complexType>
```

Table 4.7: The ontype element.

Pointcut Element

The pointcut element is shown on table 4.9. A pointcut requires a unique identifier, even though it may be declared as an anonymous pointcut, in which case an identifier is generated. It may have a collection of arguments. It has also a designator, which states the kind of primitive pointcuts and their declarations.

The primitive pointcuts are: `call`, `execution`, `get`, `set`, `handler`, `adviceexecution`, `within`, `withincode`, `this`, `args`, `target`, `cflow`, `cflowbelow`, `staticinitialization`, `initialization` and `preinitialization`. The match element describes a match for the join point specified in the pointcut. It has an identifier that identifies the target of the join point, which can either be a method execution, method call, field get, field set or exception handling. It has also the parent attribute which states the object type of the match in the case of a method execution, and indicates the object type and originating method for any of the other matches.

A pointcut may have the same modifiers as a method: `abstract`, `final`, `native`, `private`, `protected`, `public`, `static`, `synchronized` and

```
<xs:complexType name="IntertypeFieldModel">
<xs:complexContent>
  <xs:extension base="VariableModel">
   <xs:sequence>
    <xs:element name="ontype" type="OnTypeModel"
                maxOccurs="1" />
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="IntertypeMethodModel">
<xs:complexContent>
  <xs:extension base="MethodModel">
   <xs:sequence>
    <xs:element name="ontype" type="OnTypeModel"
                maxOccurs="1" />
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
```

Table 4.8: The intertype field and method elements.

`strictfp`.

Advice Element

The advice element is shown on table 4.10. Contrary to methods and pointcuts, it does not need an identifier, but rather a kind attribute, which specifies its behaviour. An advice can fall into one of the following kinds: `after`, `afterReturning`, `afterThrowing`, `around` or `before`. It has a pointcut element that links it to a pointcut that defines the location of a cross-cutting concern. It may have a collection of arguments, and may have a return type

```
<xs:complexType name="PointcutModel">
 <xs:sequence>
  <xs:element name="modifier" type="ModifierModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="argument" type="TypeModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="designator" type="DesignatorModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="match" type="MatchModel"
              minOccurs="0" maxOccurs="unbounded" />
 </xs:sequence>
 <xs:attribute name="id" type="xs:string"
               use="required" />
</xs:complexType>

<xs:complexType name="MatchModel">
 <xs:attribute name="id" type="xs:string"
               use="required" />
 <xs:attribute name="parent" type="xs:string"
                use="required" />
 <xs:attribute name="package" type="xs:string"
                use="required" />
</xs:complexType>
```

Table 4.9: The pointcut elements.

depending on the kind of advice (either `afterReturning` or `around`). If it is an `afterThrowing` advice then it may specify also exceptions to be thrown.

4.2 AspectJ Extractor

A parser was needed in order to extract the information from the source code. Our options were either to create our own parser, or extend an already exiting parser. We decided to go for the second option since it would allow

```
<xs:complexType name="AdviceModel">
 <xs:sequence>
  <xs:element name="kind" type="KindModel" maxOccurs="1" />
  <xs:element name="pointcut" type="PointcutIDModel" />
  <xs:element name="argument" type="TypeModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="return" type="TypeModel"
              maxOccurs="1" />
  <xs:element name="exception" type="TypeModel"
              maxOccurs="unbounded" />
 </xs:sequence>
</xs:complexType>
```

Table 4.10: The advice elements.

us to use a newer version of AspectJ, as well as to enable us to remain current with the newer releases of the AspectJ.

The two choices for an AspectJ compiler are the `ajc` which is the standard AspectJ compiler [Pro], and the AspectBench Compiler [Com] which is a complete re-implementation of AspectJ.

We chose the `ajc` compiler due to its support for the Eclipse IDE. Our AspectJ Extractor is integrated into Eclipse in the form of a plugin. An AspectJ application needs to be compiled and then have the aspects woven into it, our approach is similar. The AspectJ Extractor parses the source code and retrieves the static information from the classes. Because we are not compiling the source code, the weaver will not weave the aspects but the relationships will remain. We use the relationship information from the weaver to get the join point match for the pointcuts.

When parsing, also anonymous and inner classes and aspects are counted as classes and aspects inside a package, therefore the number of classes and

aspects may easily outnumber the number of physical java and aspectj files.

We extended the ajc compiler to create its own AST and retrieve the relationships from the weaver (the weaver knows where everything fits). After the AST is done and the relations are stored, match the two into a XML file, following the XML Schema.

4.3 Analysis Using XQuery

XQuery is a structured query language designed to query XML documents. It is based on a tree-structured representation of the data contained inside the XML document. It has some semantic similarities to SQL and is a flexible solution for a non-standard database.

XQuery defines the FLWOR expression which stands for: `FOR LET WHERE ORDERBY RETURN`. The FLWOR expression is the basis for queries in XQuery. The basic structure of a query in XQuery is shown on Table 4.11.

```
for $foo in /path/to/node
let $bar := $foo/child/grandchild
where ($foo=$bar) or ($foo!=$bar)
order by ...
return element result {
  attribute attfoo{$foo/@fooattribute},
  attribute attbar{$bar/@barattribute}
}
```

Table 4.11: XQuery example.

Both `$foo` and `$bar` are variables. `$foo` is of "type" `node` and `$bar` is of "type" grandchild.

4.3.1 Analysis

System Model Queries

In this subsection we present the XQuery version of some system queries defined in Section 3.2.1.

Find all subpackages of a package: Find all packages declared within the scope of a package with id *parent.package*.

```
let $p := "parent.package"
for $a in /system/package[@id!=$p]
where contains($a/@id,$p)
return element pacakge
{
  attribute id{$p},
  attribute contains{$a/@id}
}
```

Aspect and Class Model Queries

In this subsection we present the XQuery version of some aspect and class queries defined in Section 3.2.1.

Find all variables and elements in a class: For each class in the system, find the variables and methods inside that class.

```
for $class in /system/package/class
return element class
{
  attribute id{$class/@id},
  for $v in $class/variable
  return element variable {
    attribute id{$v/@id}
  },
  for $m in $class/method
  return element method {
    attribute id{$m/@id}
  }
```

```
}
```

Find all pointcuts in an aspect: For all aspects in the system, find the pointcuts inside that aspect.

```
for $asp in /system/package/aspect
return element aspect  {
 attribute id{$asp/@id},
 for $p in $asp/pointcut
 return element pointcut {
   attribute id{$p/@id}
 }
}
```

Metrics

In this subsection we present the XQuery version of some metrics queries defined in Section 3.2.2.

Cross-cutting vocabulary size:

```
count (/system/package/aspect)
```

Weighted Operations per Component:

```
for $a in /system/package/aspect
return element aspect
{
  attribute id{$a/@id},
  for $b in $a/advice
  let $na := count($b/argument)
  return element advice {
    attribute kind{$b/kind/@id},
    attribute refersto{$b/pointcut/@id},
    attribute arguments{$na}
  }
}
```

Count the number of abstract aspects:

```
for $package in /system/package
let $na := count($package/aspect[modifier/@id="abstract"])
return element package
{
  attribute id{$package/@id},
  attribute abstract{$na}
}
```

Design Guidelines

In this subsection we present the XQuery version of some design guideline queries defined in Section 3.2.3.

Find all pointcuts using a cflow primitive: For each aspect in the system, find the aspects with a pointcut using the cflow primitive pointcut.

```
for $pointcut in /system/package/aspect/pointcut
where ($pointcut/designator/@id="cflow")
return element aspect
{
  attribute id{$pointcut/../@id},
  attribute pointcut{$pointcut/@id}
}
```

Abstract aspects with concrete pointcuts: For each aspect in the system, if the aspect is abstract, find all aspects that extend it.

```
for $a in /system/package/aspect
where $a/modifier/@id="abstract"
return element aspect {
   attribute id{$a/@id},
   for $p in $a/pointcut
   where not(exists($p/modifier[@id="abstract"]))
   return element pointcut {
     attribute id{$p/@id}
  }
}
```

Chapter 5

Case Studies

Our approach has been evaluated through five case studies. All five systems are open source, with the source code available over the internet. The analysis performed on these systems is comprised of the metrics queries defined in section 3.2.2 and of some design guideline queries defined in section 3.2.3. The goal of these case studies is to demonstrate the usefulness of the query-based analysis approach.

5.1 Aspect-Oriented Systems Overview

We have made experiments involving the queries previously described in Section 3.2, which are related to measurement and analysis of aspect-oriented systems. Table 5.1 shows some size metrics and information about the systems in question.

Our integrated approach and related tool support was found very useful to help understand, measure and analyze the systems in the five case studies

we have conducted. Using the special purpose prototype tools we could understand the architecture and dependencies of the systems written in AspectJ in a very effective way. Some representative queries and results related to AJHSQLDB follow.

The first aspect-oriented system we have used id called AJHSQLDB, the result of an AOP refactoring case study [MSS06]. The AJHSQLDB version used in this case study contains the code after the refactoring of tracing, logging, profiling and exception handling.

The second one, AJHotDraw [oT], is an aspect-oriented refactoring of JHotDraw, a relatively large and well-designed open source Java framework for technical and structured 2D graphics.

The third one is Contract4J [Teab], a tool that supports Design by Contract programming in Java 5. Contract tests are defined using Java 5 annotations and aspects written in AspectJ evaluate the test expressions at runtime and handle failures.

The fourth one is DJProf [Pea] which is an experimental Java profiling tool. It uses AspectJ to insert the instrumentation for profiling instead of approaches such as the Java Machine Profiler Interface. DJProf aims to enable profiling of Java programs without source code modification and uses the Load-Time Weaving capability of AspectJ to achieve this goal.

The fifth one is the AspectJ Exception FrameWork (AJEFW) [Teaa], which provides a framework to handle exceptions focusing on core reuse for different types of exceptions.

Case Study	Size (LoC)	Classes	Aspects	Vocab. Size	AOS-KB (LoC)
AJHSQLDB	149,536	330	31	361	54,510
AJHotDraw	36,304	401	10	411	21,798
Contract4J	5,051	44	14	58	3,276
DJProf	1,124	20	6	26	1,530
AJEFW	857	25	3	28	737

Table 5.1: AspectJ systems to be analyzed

5.2 Experimental Results

In this section we analyse the case studies using a set of general analysis exploratory queries, as well as queries based on proposed metrics, design guidelines and package metrics. As a result of our analysis, we suggest possible changes to the case studies that affect the systems at the interface, component (composition and decomposition) and package level.

5.2.1 General System Analysis

Aspect-Package Dependencies

Table 5.2 shows the maximum number of packages affected by a single aspect. By comparing this number with the total of packages in the system we are able to assess the effect that those aspects have on the system. Both AJHSQLDB and AjHotDraw have aspects that affect more than 80% of the total of packages in the system. From this result it is possible to infer that both the TracingFullAspect and ReportThrows aspect are highly coupled to other the packages in the system.

Both the TracingFullAspect and CmdCheckViewRef aspects are defined inside packages created for the inclusion of aspects in the system. This re-

Case Study	Aspect	Max. Packages	Total Packages
AJHSQLDB	TracingFullAspect	13	14
AJHotDraw	CmdCheckViewRef	4	21
Contract4J	ReportThrows	8	10
DJProf	HeapAspect	1	4
AJEFW	AspectError	1	4

Table 5.2: Aspect affecting maximum number of packages.

sulted in the introduction of new dependencies between packages in the system, which refer to dependencies between the old and new packages. This does not affect any already existing dependencies, but makes the system more complex. Since Contract4J is not a refactoring of an existing application to include aspects, it is not possible to assert if the aspects introduce any new dependencies.

Aspect-Package Dependency Introductions

AJHSQLDB is a refactoring of HSQLDB to include aspects. The aspect ValuePoolingAspect is an aspect introduced into "replace" the ValuePool class. By querying the knowledge base, it was verified that the ValuePoolingAspect affects 7 packages in total while the ValuePool class affects only 2 packages, therefore there are 5 extra dependencies being introduced. This aspect is comprised of 11 pointcuts. Each of these pointcuts targets execution of the constructor for some primitive wrapper classes, and relates to an around advice. In summary, this aspects checks if the value being placed inside the wrapper class has the same value as another one already in the pool. The aspect then intercepts this execution with the use of the around advice and returns a reference to the object in the pool. By using an

aspect-oriented approach, it was possible to apply the value pooling to the entire system in a fairly easy way, and oblivious to the component program (which, for all purposes, still calls the constructor for the wrapper class).

Unfortunately, this refactoring does not behave in the same manner as the refactored portion of the program. It was possible to determine this behaviour through our analysis. The introduced dependencies are not part of the original system design, and could lead to inconsistencies and undesired behaviour. More knowledge about the system and functionality in question would be required in order to assert the benefits or drawbacks of this specific approach. The introduced dependencies are between the store package and the packages jdbc, resources, scriptio, util and persist.

5.2.2 Metrics

Size and Coupling Metrics

We measured the five systems using the metrics described in Section 3.2.2. Table 5.3 shows the five case studies according to the maximum measurement for the following metrics: weighted primitive pointcut, number of attributes, number of cross-cutting attributes, coupling between components. The measurements shown correspond to the highest value achieved by an entity in each of the case studies.

Coupling Between Components (CBC)

The CBC metric counts the number of times a class was used as an instance variable or method argument. Both DJProf and AJEFW use primitives

Case Study	CBC	WPP	NoA	NoCA	WOC	CCC
AJHSQLDB	8	37	635	19	5	6
AJHotDraw	4	6	35	2	1	1
Contract4J	7	7	24	4	3	2
DJProf	0	5	8	2	1	6
AJEFW	0	1	7	1	1	1

Table 5.3: Size and coupling queries.

and standard library Java objects and do not use any of their own declared objects.

Weighted Primitive Pointcuts (WPP)

The WPP metric is used to assess the complexity of a pointcut by the number of primitives it uses to compose its join point expression. A higher number for WPP indicates that the join point expression is more complex. AJHSQLDB has the most complex pointcut while the remaining case studies have somewhat manageable maximum pointcut complexity.

One way to reduce a high WPP is to use one or more interfaces in the component program to determine the location of a cross-cutting concerns. By using interfaces, some of the more related join points can be grouped together, and be more easily specified through the primitive pointcuts. The decomposition of pointcuts would result also in a lower WPP. The resulting pointcuts could then be combined to reconstruct the original set of join points.

Upon further analysis of the case studies through our query-based approach, we could determine some other characteristics for pointcuts in question, as shown on Table 5.4. There are two interesting results from this

analysis. The first one is that even though the pointcut in AJHSQLDB is the most complex one, it is not the one with the most number of join points. The second result is that the pointcut with maximum WPP in Contract4J had no matching join points. This could be due to an error in the pointcut design and further inspection would be required to determine if that is the case.

Case Study	Pointcut	WPP	Arguments	Matches
AJHSQLDB	traceFieldSets	37	0	24
AJHotDraw	commandExecute	6	0	18
Contract4J	invarTypeMethod	7	2	0
DJProf	allUses	5	0	92
AJEFW	AspectMajorGroup	1	0	16

Table 5.4: Pointcuts with highest WPP.

With our exploratory analysis, we were able to identify the pointcuts with a WPP value of 0. They are located in the AJHSQLDB, with names getNonThreadSafe and nonThreadSafe2. It was verified that its WPP values is due to the use of named pointcuts instead of primitive pointcuts. Using the getNonThreadSafe pointcut, for example, it was possible to verify that it is composed of two named pointcuts, namely voidUIMethodCalls and excludedJoinPoints. The later one is a composition of one primitive and one named pointcut, threadSafeCalls. Table 5.5 shows the WPP for each of those pointcuts. Even though the result of the WPP query states a value of 0 for getNonThreadSafe, after the analysis of its properties one can say that the real value for the WPP of getNonThreadSafe is the sum of the WPP for its composing pointcuts, which would result in a WPP of 8. Likewise, the WPP value for nonThreadSafe2 could be updated to 11.

Pointcut	WPP
voidUIMethodCalls	3
excludedJoinPoints	1
threadSafeCalls	4

Table 5.5: Pointcuts used in the pointcut getNonThreadSafe.

Number of Attributes (NoA)

The NoA metric is used to asses the complexity of a class based on its instance variables. A class with higher NoA is probably more complex that a class with lower NoA. Table 5.3 shows the maximum NoA for each of the case studies. Our query-based approach allowed us to better analyze the classes in question and we could find that, for AJHSQLDB, AJHotDraw and Contract4j, over 94% of variables had the `static` modifier and over 91% of the variables had also the `final` modifier. One possible conclusion is that such classes are being used to define constants used throughout the system. Table 5.6 shows the results of our extended analysis.

Case Study	Class	NoA	`static`	`final`
AJHSQLDB	Token	635	635	632
AJHotDraw	FigureAttributeConstant	35	33	32
Contract4J	KnownBeanKeys	24	24	24
DJProf	WasteData	4	0	0
AJEFW	Principal	7	0	0

Table 5.6: Analysis of classes with high NoA.

Number of Cross-cutting Attributes (NoCA)

The NoCA metric is used to assess the complexity of an aspect by the number of pointcuts it contains. A higher number of pointcuts may indicate

that an aspect affects more join points in a system. Table 5.7 shows the results of further analysis on those aspects with high NoCA for each of the case studies.

Case Study	Aspect	NoCA	Total Matches	Advices
AJHSQLDB	AccessControlAspect	19	19	19
AJHotDraw	SelChgdNotification[1]	2	4	2
Contract4J	AbstractConditions	4	0	0
DJProf	WasteAspect	2	116	2
AJEFW	AspectMajorGroup	1	16	0

Table 5.7: Analysis of aspects with high NoCA.

The aspects in AJHSQLDB and AJHotDraw use a higher number of pointcuts to match more specific join points. Each pointcut in AccessControlAspect matches exactly one join point and has its corresponding advice to deal with the cross-cutting concern. After performing more queries it could be verified that each pointcut deals with different combinations of arguments. A possible conclusion is that such pointcuts address the same cross-cutting concern, but the data involved in each join point match varies thus requiring a more refined pointcut evaluation.

A possible solution to this issue is to create an interface for cross-cutting concerns. Also this interface can be extended to some of the arguments being passed to the advice, and would allow for a easier handling of the data. This would result in a composition of pointcuts and advices with similar interfaces and help reduce the number of pointcuts that are required to target this cross-cutting concern. This approach would be better suited for pointcuts with low WPP, since the composition of pointcuts could lead

[1] Abbreviated from SelectionChangedNotification.

to a significant increase in WPP.

Another possibility is to decompose the aspect further, and modularize the cross-cutting concerns at a lower level of granularity. This approach would result in a increase in the number of aspects in the system, but would help to localize the cross-cutting concern inside a more consistent aspect. All the pointcuts dealing with a specific type of argument, or targeting a specific part of the system would be placed together. This approach is better suited for pointcuts with high WPP, since it would break down the pointcut into smaller units.

The aspect in Contract4J is an abstract aspect, and our query shows it is extended by 7 other aspects. Table 5.8 shows those aspects and their results for our analysis. They have no effect on the system, since they match no join points.

Aspect	Max. Matches	Advices
ConstructorBoundaryConditions	0	2
InvariantCtorConditions	0	1
InvariantFieldConditions	0	2
InvariantFieldCtorConditionsPerCtor	0	2
InvariantMethodConditions	0	1
InvariantTypeConditions	0	2
MethodBoundaryConditions	0	3

Table 5.8: Aspects that extend AbstractConditions.

Weighted Operations per Component (WOC)

The WOC metric is used to identify advices with a high number of arguments. An advice that deals with more arguments is likely to be more complex than an advice with fewer arguments. Table 5.9 shows the result

of our query on the case studies.

Case Study	Advice	Pointcut	WOC
AJHSQLDB	before	beforeUpdateSingleRow	5
AJHotDraw	after	invalidateSelFigure	1
Contract4J	around	invarSetField	3
DJProf	before	allUses	1
AJEFW	around	errorPoints	1

Table 5.9: Analysis of advices with high WOC.

One suggestion to deal with this issue is to decompose the advice and its relating pointcut into smaller parts that require less arguments.

Cross-cutting Coupling (CCC)

The CCC metric is used to assess the degree of coupling of a join point. A higher value indicates that a join point is being targeted by more pointcuts. This may lead to issues when refactoring or maintaining such join point, since a change in the join point will affect a greater number of aspects. Table 5.10 shows the results of our analysis. The (M) indicates that the join point is a method, while the (F) indicates the join point is a field (instance variable).

Case Study	Join Point	Parent	CCC
AJHSQLDB	dropColumn (M)	TableWorks	6
AJHotDraw	execute (M)	RedoCommand	1
Contract4J	isEnabled (F)	Contract4J	2
DJProf	constructor	Hashtable	6
AJEFW	io (M)	ErrorThrower	1

Table 5.10: Analysis of join points with high CCC.

In the case of AJHSQLDB, any change in the dropColumn method may

affect 6 other pointcuts and/or advices in the system. One way to address this problem is to define interfaces for the join points. This would result in the pointcuts being coupled to a single entity and allow for more flexibility in the join point construction.

In the case of DJProf, the join point is a constructor for a standard Java library class, the Hashtable. Because it is part of the standard Java library, it is more likely to remain unchanged, and can even be considered as an interface (it is part of the java.util API).

5.2.3 Design Guidelines

Around Advices

Table 5.11 shows the results of the application of the around advice design guideline query. This guideline is provided by Dufour et al. to improve the execution of the software and minimize the overhead imposed by the AspectJ compiler.

Case Study	Around Advices
AJHSQLDB	36
AJHotDraw	0
Contract4J	6
DJProf	0
AJEFW	1

Table 5.11: Design guidelines - Around advices.

The number of `around` advices may result also in poor system performance. The query show that there are some advices both in AJHSQLDB and Contract4J that could potentially be changed and result in an increased performance. The suggestion given to this guideline is to use `after returning`

advices if possible.

Other Performance Design Guidelines

Table 5.12 shows the results of the application of other design guideline queries. These guidelines are provided by Dufour et al. [DGH+03]. The table shows in order: the maximum number of join points per pointcut, the total number of `cflow` pointcuts and the total number of `pertarget` pointcuts.

Case Study	Max. Matches	`cflow`	`pertarget`
AJHSQLDB	3570	7	0
AJHotDraw	18	0	0
Contract4J	338	3	0
DJPorf	139	0	0
AJEFW	16	0	0

Table 5.12: Design guidelines - Other performance guidelines.

Each join point that is matched by a pointcut incurs overhead being placed on the software. The larger the number of maximum join points, the worse the performance the system will suffer. We can assess from the results of our query that both AJHSQLDB and Contract4J have a pointcut which introduces significant amounts of overhead in the system.

The use of `cflow` pointcuts should be kept to the absolute minimum, according to Dufour et al., and the use of `withincode` pointcuts is preferred. Our query-based approach allowed us to identify the use of `cflow` pointcuts in the system, as shown on Table 5.12. Even though the number of instances may be small when compared to the rest of the system, it is a valid point of interest. None of the systems in this case studies had a `pertarget` pointcut.

Dead Code Guidelines

This guideline is based on the notion that an `around` advice may block another advice that targets the same join point. This behaviour may be corrected by the use of the `proceed` command. Table 5.13 shows the result of our query-based analysis. The query returns all instances where an `around` advice targets the same join point as any other advice in the system. Code inspection would be required in order to assert the use of `proceed`.

None of our case studies presented the possibility of having dead code.

Case Study	Dead Code
AJHSQLDB	0
AJHotDraw	0
Contract4J	0
DJProf	0
AJEFW	0

Table 5.13: Design guidelines - Dead code detection.

Redundant Execution Guidelines

This guideline is based on the notion that if there are concrete pointcuts inside an abstract aspect, the advice related to each of these pointcuts is executed $n + 1$ times, where n is the number of concrete instances of the abstract aspect. Table 5.14 shows the number of concrete pointcuts inside abstract packages.

5.2.4 Package Metrics

In the following sections we show the measurement of the package dependency metrics for AJHSQLDB, AJHotDraw and Contract4J case studies.

Case Study	Possible Redundancy
AJHSQLDB	2
AJHotDraw	0
Contract4J	4
DJProf	0
AJEFW	0

Table 5.14: Design guidelines - Redudant execution.

For completeness, the package dependency metrics for DJProf and AJEFW are present in Table 5.18 and Table 5.19, respectivelly. The metrics are used as described in Section 3.2.2.

AJHSQLDB Package Dependency Metrics

Packages	ABS	CE	CA	I
org.hsqldb.aspects	0.3	159	0	1
org.hsqldb	0.081	4986	117	0.977
org.hsqldb.index	1	3	12	0.2
org.hsqldb.jdbc	0	1398	2	0.998
org.hsqldb.lib	0.190	1103	199	0.847
org.hsqldb.lib.java	0	25	0	1
org.hsqldb.persist	0.1	632	36	0.946
org.hsqldb.resources	0	20	0	1
org.hsqldb.rowio	0.333	473	46	0.911
org.hsqldb.sample	0	81	0	1
org.hsqldb.scriptio	0.25	221	1	0.995
org.hsqldb.store	0	259	9	0.966
org.hsqldb.types	0	14	12	0.538
org.hsqldb.util	0.030	1733	1	0.999

Table 5.15: AJHSQLDB package dependency metrics.

The dependency metrics show that the majority of the packages have high instability, with only the index and types packages having instability significantly lower than the others. One interesting result is the analysis

of the org.hsqldb.aspects package, which has no afferent coupling. In other words, the component program is completely oblivious to the aspect program, and no extra outbound dependency was introduced between the packages appart from the aspects package. Table 5.15 shows the results of our package analysis.

AJHotDraw Package Dependency Metrics

The dependency metrics for this case study show two interesting factors about AJHotDraw. The first is the abstract nature of the framework package, which is the core of the JHotDraw application as stated in Section 5.1. The other is related also to the framework package which is the most stable of all, since the entire system is built on top of this framework. Table 5.16 shows the results of our package analysis.

Contract4J Package Metrics

The Contract4J package metrics show that the abstract nature of the system is well spread among the system. They show also no package as having significant more stability than others. This is, in part, due to the purpose of the system, which is described in Section 5.1. It's purpose is to server as a library, supporting design by contract development. Table 5.17 shows the results of our package analysis.

Packages	ABS	CE	CA	I
org.jhotdraw.applet	0	89	4	0.956
org.jhotdraw.application	0	173	17	0.910
org.jhotdraw.cmdcontracts	0	5	0	1
org.jhotdraw.contrib	0.1	1023	26	0.975
org.jhotdraw.contrib.dnd	0.2	172	4	0.977
org.jhotdraw.contrib.html	0.289	368	0	1
org.jhotdraw.contrib.zoom	0.076	355	0	1
org.jhotdraw.figures	0.065	921	16	0.982
org.jhotdraw.framework	0.807	368	1123	0.246
org.jhotdraw.lib	1	0	1	0
org.jhotdraw.observselection	0.25	52	0	1
org.jhotdraw.persistence	0	30	0	1
org.jhotdraw.samples.javadraw	0	165	2	0.988
org.jhotdraw.samples.minimap	0	10	0	1
org.jhotdraw.samples.net	0	33	0	1
org.jhotdraw.samples.nothing	0	8	0	1
org.jhotdraw.samples.pert	0	83	0	1
org.jhotdraw.standard	0.135	1873	108	0.944
org.jhotdraw.util	0.255	671	245	0.756
org.jhotdraw.util.collections.jdk11	0	112	0	1
org.jhotdraw.util.collections.jdk12	0	12	0	1

Table 5.16: AJHotdraw package dependency metrics.

Packages	ABS	CE	CA	I
org.contract4j5.aspects	0.214	186	0	1
org.contract4j5.configurator	0.333	90	4	0.957
org.contract4j5	0.454	106	92	0.535
org.contract4j5.enforcer	0.5	36	6	0.857
org.contract4j5.interpreter	0.4	115	30	0.793
org.contract4j5.policies	0	5	0	1
org.contract4j5.testexpression	0.363	135	36	0.789
org.contract4j5.util.debug	0	7	0	1
org.contract4j5.util	0	10	0	1
org.contract4j5.util.reporter	0.333	53	11	0.828

Table 5.17: Contract4J package dependency metrics.

Packages	ABS	CE	CA	I
djprof	0.285	52	20	0.722
djprof.timing	0.5	5	1	0.833
com.ibm.DJProf.CallCountProfiler	0	15	0	1
default	0	163	0	1

Table 5.18: DJProf package dependency metrics.

Packages	ABS	CE	CA	I
gj.ajframework	0.555	77	8	0.905
gj.ajmod	0	2	0	1
gj.app	0.1	45	0	1
gj.errormod	0	56	0	1

Table 5.19: AJEFW package dependency metrics.

Chapter 6

Conclusion And Future Work

6.1 Conclusion

In this thesis we have shown that by extending the metrics and analysis capabilities of current approaches, which are often restricted to code-level evaluations, we can define a more comprehensive approach for the analysis of aspect oriented systems based on higher-level analysis and metrics, a set of criteria (e.g., dead code, redudant execution, dependency constraints) for the analysis and extended metric sets (e.g., subsystem dependency, interface level separation of concerns, architectural coupling) suitable for a higher-level approach.

We implemented tool support for our approach, which combines a parser for Java/AspectJ and a XQuery interface for the Aspect Knowledge Base. The parser for Java/AspectJ uses the work already done in AJDT, and should provide for an easier update for later versions of AspectJ. The Aspect Knowledge Base is implemented in XML and provides great flexibility and

adaptability.

We provided experimental support based on five relevant case studies. The results of the analysis indicated the usefulness and impact of the approach in understanding and assessing the quality of aspect-oriented software systems. Overall we believe the techniques proposed in this thesis have a positive impact on aspect-oriented programming and contribute to the field of software engineering.

6.2 Future Work

The approach and work presented in this thesis can be extended in different areas.

6.2.1 Additional Metrics and Metrics Validation

New aspect-oriented metrics have been proposed in the literature [MB, MB06, NP], as well as already proposed metrics, such as the Separation of Concern metrics defined by [CSS03], can be included in the Aspect Knowledge Base. Such metrics can be incorporated into the knowledge base by the extension of the XML Schema and use of XSL transformations. This would enable the results from other tool sets to be introduced into our knowledge base and enable a more thorough assessment of the aspect-oriented system based on combined metrics.

6.2.2 Update of Model and Tool Support

Further work can be done in the model and tool, to support the latest version of the AspectJ language, AspectJ 5.0, which has new features in par with Java 5.0. Such features include support for generic types, autoboxing and unboxing and annotations among others. As AspectJ evolves, its capabilities to encapsulate cross-cutting concerns clearly and more easily improves, by providing better constructs and syntax to deal with specific issues. Such additions would have to be correctly parsed and represented in the Aspect Knowledge Base, therefore an updated version of the parser and the software system model needs to be created.

Bibliography

[ABUoC] San Diego Aspect Browser University of California. http://www.cse.ucsd.edu/ wgg/software/ab/.

[aMS] aopmetrics Michal Stochmialek. http://aopmetrics.tigris.org/.

[Apo] Apostle. http://www.cs.ubc.ca/labs/spl/projects/apostle/.

[Aspa] AspectC++. http://www.aspectc.org/.

[Aspb] AspectC. http://www.cs.ubc.ca/labs/spl/projects/aspectc.html.

[CK94] S. Chidamber and C. Kemerer. A metrics suite for object-oriented design. *Transactions on Software Engineering*, 20(6):476–493, 1994.

[Coe] Active Aspect Wesley Coelho. http://www.cs.ubc.ca/labs/spl/projects/activeaspect/.

[Com] AspectBench Compiler. http://abc.comlab.ox.ac.uk/introduction.

[CSS03] C. Chavez C. Lucena C. Sant'Anna, A. Garcia and A. Staa. On the reuse and maintenance of aspect-oriented software: An assessment framework, 2003.

[CZ02] H. Jacobsen C. Zhang. Quantifying aspects in middleware platforms. International AOSD'02 Conference, 2002.

[CZJ] T. Javeed C. Wong C. Zhang, H. Shi and H. Jacobsen. http://www.eecg.toronto.edu/%7ejacobsen/prism/.

[DGH+03] B. Dufour, C. Goard, L. Hendren, C. Verbrugge, O. de Moor, and G. Sittampalam. Measuring the dynamic behaviour of aspectj programs, 2003.

[FF00] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. Workshop on Advanced Separation of Concerns, OOPSLA 2000, 2000.

[GK97] A. Mendhekar C. Maeda C. Lopes J. Loingtier J. Irwin G. Kiczales, J. Lamping. Aspect-oriented programming. In *Proceedings of ECOOP*, 1997.

[Han] Aspect Mining Tool J. Hannemann. http://www.cs.ubc.ca/j̃an/amt/.

[J] Hyper J. http://www.alphaworks.ibm.com/tech/hyperj.

[JGB06] M. Badri J. Gelinas and L. Badri. Cohesion measure for aspects. *Journal of Object Technology*, 5(7):97–114, 2006.

[Kic03] G. Kiczales. Aspect-oriented programming - the fun has just begun. Key note address of 2nd International Conference on Aspect-Oriented Software Development, 2003.

[KLO01] D. Orleans K. Lieberherr and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Communications of the ACM*, 44(10):39–41, 2001.

[Lee] Ken Wing Kuen Lee. Introduction to aspect oriented programming. The Hong Kong University of Science and Technology.

[LOO] LOOM.NET. http://www.ajlopez.net/itemve.php?id=996.

[Mar94] Robert Martin. Oo design quality metrics. an analysis of dependencies, 1994.

[MB] Rachel Harrison Marc Bartsch. An evaluation of coupling measures for aspectj. In *Workshop on Linking Aspect Technology and Evolution (LATE) 2006 in Conjunction with AOSD 2006*.

[MB06] Rachel Harrison Marc Bartsch. A coupling framework for aspectj. In *Proceedings of the 10th Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2006.

[MK03a] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of ECOOP*, 2003.

[MK03b] H. Masuhara and G. Kiczales. A modeling framework for aspect-oriented mechanisms. In *Proceedings of ECOOP2003*, pages 2–28, 2003.

[MSS06] U. Eibauer M. Storzer and S. Schoeffmann. Aspect mining for aspect refactoring: An experience report. Workshop TEAM06, held in conjunction with ECOOP06, 2006.

[NP] Zoltan Porkolab Norbert Pataki, Adam Sipos. Measuring the complexity of aspect-oriented programs with multiparadigm metric. In *ECOOP 2006 Doctoral Symposium and PhD Workshop*.

[oT] AjHotDraw Delft University of Technology. http://ajhotdraw.sourceforge.net/.

[Pea] David J. Pearce. http://www.mcs.vuw.ac.nz/ djp/djprof/.

[Pro] AspectJ Eclipse AJDT Project. http://www.eclipse.org/aspectj.

[RV] R. Rajagopolan and K.D. Volder. Qjbrowser: A query-based approach to explore crosscutting concerns. submitted for publication.

[Sha] AspectC Sharp. http://www.dsg.cs.tcd.ie/dynamic/?category_id=168.

[SP04] D. Shepherd and L.L. Pollock. Ophir:a framework for automatic mining and refactoring of aspects. Technical Report 2004-3, University of Delaware, 2004.

[Teaa] AJEFW Team. http://sourceforge.net/projects/ajefw/.

[Teab] Contract4J Team. http://www.contract4j.org/contract4j.

[TEO01] G. Kiczales K. Lieberherr T. Elrad, M. Aksit and H. Ossher. Discussing aspects of aop. *Communications of the ACM*, 44(10):33–38, 2001.

[W3C] XML Schema W3C. http://www.w3.org/xml/schema.

[Wea] Weave.NET. http://www.dsg.cs.tcd.ie/dynamic/?category_id=193.

[Zha] J. Zhao. Towards a metrics suite for aspect-oriented software.

[Zha04] J. Zhao. Measuring coupling in aspect-oriented systems, 2004.

[ZX04] J. Zhao and B. Xu. Measuring aspect cohesion, 2004.

Appendix A

AspectJ Quick Reference

A.1 Aspects

```
aspect A {  }
  defines the aspect A
privileged aspect A {  }
  A can access private fields and methods
aspect A extends B implements I, J {  }
  B is a class or abstract aspect, I and J are interfaces
aspect A percflow( call(void Foo.m()) ) {  }
  an instance of A is instantiated for every control flow through
  calls to m()

general form:
  [ privileged ] [ Modifiers ] aspect Id
    [ extends Type ] [ implements TypeList ] [ PerClause ]
    { Body }

where PerClause is one of:
  pertarget ( Pointcut )
  perthis ( Pointcut )
  percflow ( Pointcut )
  percflowbelow ( Pointcut )
  issingleton ()
```

A.2 Pointcut Definitions

```
private pointcut pc() : call(void Foo.m()) ;
  a pointcut visible only from the defining type
pointcut pc(int i) : set(int Foo.x) && args(i) ;
  a package-visible pointcut that exposes an int.
public abstract pointcut pc() ;
  an abstract pointcut that can be referred to from anywhere.
abstract pointcut pc(Object o) ;
  an abstract pointcut visible from the defining package. Any
  pointcut that implements this must expose an Object.

general form:
  abstract [Modifiers] pointcut Id ( Formals ) ;
  [Modifiers] pointcut Id ( Formals ) : Pointcut ;
```

A.3 Advice Declarations

```
before () : get(int Foo.y) { ... }
  runs before reading the field int Foo.y
after () returning : call(int Foo.m(int)) { ... }
  runs after calls to int Foo.m(int) that return normally
after () returning (int x) : call(int Foo.m(int)) { ... }
  same, but the return value is named x in the body
after () throwing : call(int Foo.m(int)) { ... }
  runs after calls to m that exit abruptly by throwing an exception
after () throwing (NotFoundException e) : call(int Foo.m(int)) { ... }
  runs after calls to m that exit abruptly by throwing a
  NotFoundException. The exception is named e in the body
after () : call(int Foo.m(int)) { ... }
  runs after calls to m regardless of how they exit
before(int i) : set(int Foo.x) && args(i) { ... }
  runs before field assignment to int Foo.x. The value to be
  assigned is named i in the body
before(Object o) : set(* Foo.*) && args(o) { ... }
  runs before field assignment to any field of Foo. The value to be
  assigned is converted to an object type (int to Integer, for
  example) and named o in the body
int around () : call(int Foo.m(int)) { ... }
```

```
  runs instead of calls to int Foo.m(int), and returns an int. In the
  body, continue the call by using proceed(), which has the same
  signature as the around advice.
int around () throws IOException : call(int Foo.m(int)) { ... }
  same, but the body is allowed to throw IOException
Object around () : call(int Foo.m(int)) { ... }
  same, but the value of proceed() is converted to an Integer, and
  the body should also return an Integer which will be converted
  into an int

general form:
  [ strictfp ] AdviceSpec [ throws TypeList ] : Pointcut { Body }

where AdviceSpec is one of
  before ( Formals )
  after ( Formals )
  after ( Formals ) returning [ ( Formal ) ]
  after ( Formals ) throwing [ ( Formal ) ]
  Type around ( Formals )
```

A.4 Special Forms

```
thisJoinPoint
  reflective information about the join point.
thisJoinPointStaticPart
  the equivalent of thisJoinPoint.getStaticPart(), but may use
  fewer resources.
thisEnclosingJoinPointStaticPart
  the static part of the join point enclosing this one.
proceed ( Arguments )
  only available in around advice. The Arguments must be the
  same number and type as the parameters of the advice.
```

A.5 Intertype Member Declarations

```
int Foo . m ( int i ) { ... }
  a method int m(int) owned by Foo, visible anywhere in the
  defining package. In the body, this refers to the instance of Foo,
  not the aspect.
```

```
private int Foo . m ( int i ) throws IOException { ... }
  a method int m(int) that is declared to throw IOException, only
  visible in the defining aspect. In the body, this refers to the
  instance of Foo, not the aspect.
abstract int Foo . m ( int i ) ;
  an abstract method int m(int) owned by Foo
Point . new ( int x, int y ) { ... }
  a constructor owned by Point. In the body, this refers to the new
  Point, not the aspect.
private static int Point . x ;
  a static int field named x owned by Point and visible only in the
  declaring aspect
private int Point . x = foo() ;
  a non-static field initialized to the result of calling foo(). In the
  initializer, this refers to the instance of Foo, not the aspect.

general form:
  [ Modifiers ] Type Type . Id ( Formals )
  [ throws TypeList ] { Body }
  abstract [ Modifiers ] Type Type . Id ( Formals )
  [ throws TypeList ] ;
  [ Modifiers ] Type . new ( Formals )
  [ throws TypeList ] { Body }
  [ Modifiers ] Type Type . Id [ = Expression ] ;
```

A.6 Other Inter-type Declarations

```
declare parents : C extends D;
  declares that the superclass of C is D. This is only legal if D is
  declared to extend the original superclass of C.
declare parents : C implements I, J ;
  C implements I and J
declare warning : set(* Point.*) && !within(Point) : "bad set" ;
  the compiler warns "bad set" if it finds a set to any field of
  Point outside of the code for Point
declare error : call(Singleton.new(..)) : "bad construction" ;
  the compiler signals an error "bad construction" if it finds a call
  to any constructor of Singleton
declare soft : IOException : execution(Foo.new(..));
```

```
  any IOException thrown from executions of the constructors of
  Foo are wrapped in org.aspectj.SoftException
declare precedence : Security, Logging, * ;
  at each join point, advice from Security has precedence over
  advice from Logging, which has precedence over other advice.

general form
  declare parents : TypePat extends Type ;
  declare parents : TypePat implements TypeList ;
  declare warning : Pointcut : String ;
  declare error : Pointcut : String ;
  declare soft : Type : Pointcut ;
  declare precedence : TypePatList ;
```

A.7 Primitive Pointcuts

```
call ( void Foo.m(int) )
  a call to the method void Foo.m(int)
call ( Foo.new(..) )
  a call to any constructor of Foo
execution ( * Foo.*(..) throws IOException )
  the execution of any method of Foo that is declared to throw
  IOException
execution ( !public Foo .new(..) )
  the execution of any non-public constructor of Foo
initialization ( Foo.new(int) )
  the initialization of any Foo object that is started with the
  constructor Foo(int)
preinitialization ( Foo.new(int) )
  the pre-initialization (before the super constructor is called) that
  is started with the constructor Foo(int)
staticinitialization( Foo )
  when the type Foo is initialized, after loading
get ( int Point.x )
  when int Point.x is read
set ( !private * Point.* )
  when any non-private field of Point is assigned
handler ( IOException+ )
  when an IOException or its subtype is handled with a catch block
```

```
adviceexecution()
  the execution of all advice bodies
within ( com.bigboxco.* )
  any join point where the associated code is defined in the
  package com.bigboxco
withincode ( void Figure.move() )
  any join point where the associated code is defined in the method
  void Figure.move()
withincode ( com.bigboxco.*.new(..) )
  any join point where the associated code is defined in any
  constructor in the package com.bigoxco.
cflow ( call(void Figure.move()) )
  any join point in the control flow of each call to void
  Figure.move(). This includes the call itself.
cflowbelow ( call(void Figure.move()) )
  any join point below the control flow of each call to void
  Figure.move(). This does not include the call.
if ( Tracing.isEnabled() )
  any join point where Tracing.isEnabled() is true. The boolean
  expression used can only access static members, variables bound
  in the same pointcut, and thisJoinPoint forms.
this ( Point )
  any join point where the currently executing object is an instance
  of Point
target ( java.io.InputPort )
  any join point where the target object is an instance of
  java.io.InputPort
args ( java.io.InputPort, int )
  any join point where there are two arguments, the first an
  instance of java.io.InputPort, and the second an int
args ( *, int )
  any join point where there are two arguments, the second of
  which is an int.
args ( short, .., short )
  any join point with at least two arguments, the first and last of
  which are shorts

Note: any position in this, target, and args can be replaced with a
variable bound in the advice or pointcut.
```

```
general form:
  call(MethodPat)
  call(ConstructorPat)
  execution(MethodPat)
  execution(ConstructorPat)
  initialization(ConstructorPat)
  preinitialization(ConstructorPat)
  staticinitialization(TypePat)
  get(FieldPat)
  set(FieldPat)
  handler(TypePat)
  adviceexecution()
  within(TypePat)
  withincode(MethodPat)
  withincode(ConstructorPat)
  cflow(Pointcut)
  cflowbelow(Pointcut)
  if(Expression)
  this(Type | Var)
  target(Type | Var)
  args(Type | Var , )

where MethodPat is:
  [ModifiersPat] TypePat [TypePat . ] IdPat ( TypePat | ..,  )
  [ throws ThrowsPat ]

ConstructorPat is:
  [ModifiersPat ] [TypePat . ] new ( TypePat | .. , )
  [ throws ThrowsPat ]

FieldPat is:
  [ModifiersPat] TypePat [TypePat . ] IdPat
  TypePat is one of:
  IdPat [ + ] [ []  ]
  ! TypePat
  TypePat && TypePat
  TypePat || TypePat
  ( TypePat )
```

Appendix B

Software System Schema

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
     elementFormDefault="qualified">

<xs:element name="system" type="SoftwareSystemModel" />
<xs:complexType name="SoftwareSystemModel">
 <xs:sequence>
  <xs:element name="package" type="PackageModel"
              minOccurs="1" maxOccurs="unbounded" />
 </xs:sequence>
</xs:complexType>

<xs:complexType name="PackageModel">
 <xs:sequence>
  <xs:element name="class" type="ClassModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="aspect" type="AspectModel"
              minOccurs="0" maxOccurs="unbounded" />
 </xs:sequence>
 <xs:attribute name="id" type="xs:string"
               use="required" />
</xs:complexType>

<xs:complexType name="ClassModel">
 <xs:sequence>
  <xs:element name="modifier" type="ModifierModel"
```

```
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="extends" type="ExtendsModel"
              minOccurs="0" maxOccurs="1" />
  <xs:element name="implements" type="ImplementsModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="variable" type="VariableModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="constructor" type="ConstructorModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="method" type="MethodModel"
              minOccurs="0" maxOccurs="unbounded" />
 </xs:sequence>
 <xs:attribute name="id" type="xs:string"
               use="required" />
</xs:complexType>

<xs:complexType name="AspectModel">
 <xs:complexContent>
  <xs:extension base="ClassModel">
   <xs:sequence>
    <xs:element name="declare" type="DeclareModel"
                minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="intertypefield"
                type="IntertypeFieldModel"
                minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="intertypemethod"
                type="IntertypeMethodModel"
                minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="intertypeconstructor"
                type="IntertypeConstructorModel"
                minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="pointcut" type="PointcutModel"
                minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="advice" type="AdviceModel"
                minOccurs="0" maxOccurs="unbounded" />
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
```

```
<xs:complexType name="ExtendsModel">
 <xs:attribute name="class" type="xs:string"
               use="required" />
 <xs:attribute name="package" type="xs:string"
               use="required" />
</xs:complexType>
<xs:complexType name="ImplementsModel">
 <xs:attribute name="interface" type="xs:string"
               use="required" />
 <xs:attribute name="package" type="xs:string"
               use="required" />
</xs:complexType>

<xs:complexType name="VariableModel">
 <xs:sequence>
  <xs:element name="modifier" type="ModifierModel"
              maxOccurs="unbounded" />
  <xs:element name="type" type="TypeModel" maxOccurs="1" />
 </xs:sequence>
 <xs:attribute name="id" type="xs:string"
               use="required" />
</xs:complexType>

<xs:complexType name="ConstructorModel">
 <xs:sequence>
  <xs:element name="modifier" type="ModifierModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="argument" type="TypeModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="throws" type="ThrowsModel"
              minOccurs="0" maxOccurs="unbounded" />
 </xs:sequence>
</xs:complexType>

<xs:complexType name="MethodModel">
 <xs:complexContent>
  <xs:extension base="ConstructorModel">
   <xs:sequence>
    <xs:element name="return" type="TypeModel" maxOccurs="1" />
   </xs:sequence>
```

```
   <xs:attribute name="id" type="xs:string"
                 use="required" />
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="PointcutModel">
 <xs:sequence>
  <xs:element name="modifier" type="ModifierModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="argument" type="TypeModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="designator" type="DesignatorModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="match" type="MatchModel"
              minOccurs="0" maxOccurs="unbounded" />
 </xs:sequence>
</xs:complexType>

<xs:complexType name="AdviceModel">
 <xs:sequence>
  <xs:element name="kind" type="KindModel" maxOccurs="1" />
  <xs:element name="pointcut" type="PointcutIDModel" />
  <xs:element name="argument" type="TypeModel"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="return" type="TypeModel" maxOccurs="1" />
  <xs:element name="exception" type="ThrowsModel"
              maxOccurs="unbounded" />
 </xs:sequence>
</xs:complexType>

<xs:complexType name="IntertypeFieldModel">
<xs:complexContent>
  <xs:extension base="VariableModel">
   <xs:sequence>
    <xs:element name="ontype" type="OnTypeModel" maxOccurs="1" />
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
```

```
<xs:complexType name="IntertypeMethodModel">
<xs:complexContent>
  <xs:extension base="MethodModel">
   <xs:sequence>
    <xs:element name="ontype" type="OnTypeModel" maxOccurs="1" />
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="IntertypeConstructorModel">
<xs:complexContent>
  <xs:extension base="ConstructorModel">
   <xs:sequence>
    <xs:element name="ontype" type="OnTypeModel" maxOccurs="1" />
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="OnTypeModel">
 <xs:attribute name="id" type="xs:string"
               use="required" />
 <xs:attribute name="package" type="xs:string"
               use="required" />
</xs:complexType>

<xs:complexType name="DeclareModel">
 <xs:sequence>
  <xs:element name="declaration" type="xs:string"
              minOccurs="0" maxOccurs="unbounded" />
  <xs:element name="extends" type="ExtendsModel"
              minOccurs="0" maxOccurs="1" />
  <xs:element name="implements" type="ImplementsModel"
              minOccurs="0" maxOccurs="unbounded" />
 </xs:sequence>
 <xs:attribute name="id" type="xs:string"
               use="required" />
</xs:complexType>
```

```
<xs:complexType name="DesignatorModel">
 <xs:sequence>
  <xs:element name="target" type="xs:string" maxOccurs="1" />
 </xs:sequence>
 <xs:attribute name="id" type="xs:string"
               use="required" />
</xs:complexType>

<xs:complexType name="ModifierModel">
 <xs:attribute name="id" type="xs:string"
               use="required" />
</xs:complexType>

<xs:complexType name="TypeModel">
 <xs:attribute name="id" type="xs:string"
               use="required" />
 <xs:attribute name="package" type="xs:string"
               use="required" />
</xs:complexType>

<xs:complexType name="ThrowsModel">
 <xs:attribute name="id" type="xs:string"
               use="required" />
 <xs:attribute name="package" type="xs:string"
               use="required" />
</xs:complexType>

<xs:complexType name="KindModel">
 <xs:attribute name="id" type="xs:string"
               use="required" />
</xs:complexType>

<xs:complexType name="MatchModel">
 <xs:attribute name="id" type="xs:string"
               use="required" />
 <xs:attribute name="parent" type="xs:string"
               use="required" />
 <xs:attribute name="package" type="xs:string"
               use="required" />
```

```
</xs:complexType>

<xs:complexType name="PointcutIDModel">
 <xs:attribute name="id" type="xs:string"
               use="required" />
</xs:complexType>

</xs:schema>
```