Measuring and Characterizing Crosscutting in Aspect-Based Programs: Basic Metrics and Case Studies

Roberto E. Lopez-Herrejon[1] and Sven Apel[2]

[1] Computing Laboratory, University of Oxford, England
[2] School of Computer Science, University of Magdeburg, Germany
rlopez@comlab.ox.ac.uk, apel@iti.cs.uni-magdeburg.de

Abstract. Aspects are defined as well-modularized crosscutting concerns. Despite being a core tenet of Aspect Oriented Programming, little research has been done in characterizing and measuring crosscutting concerns. Some of the issues that have not been fully explored are: What kinds of crosscutting concerns exist? What language constructs do they use? And what is the impact of crosscutting in actual Aspect Oriented programs? In this paper we present basic code metrics that categorize crosscutting according to the number of classes crosscut and the language constructs used. We applied the metrics to four non-trivial open source programs implemented in AspectJ. We found that for these systems, the number of classes crosscut by advice per crosscutting is small in relation to the number of classes in the program. We argue why we believe this result is not atypical for Aspect Oriented programs and draw a relation to other non-AOP techniques that provide crosscutting.

1 Introduction

Aspects are defined as well-modularized crosscutting concerns, that is, concerns whose implementation would usually involve (crosscut) multiple traditional modular units such as classes. Despite the increasing interest and research in *Aspect Oriented Programming (AOP)*, very little attention has been paid to measuring and characterizing crosscutting in actual programs [8].

In this paper we present a set of basic code metrics that categorize crosscutting according to the number of classes crosscut and their language constructs. To facilitate the description, we present them semi-formally using a functional programming style. Our metrics rate a crosscutting within a spectrum that goes from *homogeneous* to *heterogeneous*, depending on the number of classes crosscut by pieces of advice in relation to the number of classes crosscut by all crosscutting mechanisms of AspectJ. This distinction helps drawing a relation with other technologies that also provide support for crosscutting [7].

By analysing actual programs and categorizing their crosscutting, our metrics shed light on the impact of aspects on the overall program structure. We applied our metrics to four non-trivial AspectJ programs. We found that for these programs, the number of classes crosscut by advice per crosscutting is small in relation to the

M.B. Dwyer and A. Lopes (Eds.): FASE 2007, LNCS 4422, pp. 423–437, 2007.

number of classes in the program. We argue why we believe this result is not atypical for Aspect Oriented programs and draw a relation to other non-AOP techniques that provide crosscutting.

2 Aspect Oriented Programming

Aspect Oriented Programming (AOP) is a novel software development paradigm that aims at modularizing *aspects*, which are defined as well-modularized crosscutting concerns [10][25]. This type of concerns cuts across traditional module boundaries such as classes and interfaces, and their implementation is scattered and tangled with the implementation of other concerns. *AspectJ* is the flagship language of AOP [10]. This is the implementation language of the case studies we evaluated, thus we use AspectJ to illustrate and define our metrics. This section explains the basic constructs of the language. In AspectJ, an application consists of two parts: *base code* which corresponds to standard Java classes and interfaces, and *aspect code* which contains the crosscutting code. Next we describe the two types of crosscuts that AspectJ provides.

2.1 Static Crosscuts

Static crosscuts affect the static structure of a program [26]. We consider *Inter-Type Declarations (ITDs)*, also known as *introductions*, that add fields, methods, and constructors to existing classes and interfaces[1]. Let us consider the example in Figure 1a. It contains an aspect `A` and three classes `X`, `Y`, and `Z`. The symbols s_i stand for any statement. Aspect `A` has four ITDs that introduce: 1) field `q` in class `X`, 2) method `n` in class `Y`, 3) constructor for class `Z`, and 4) method `foo` to class `X`.

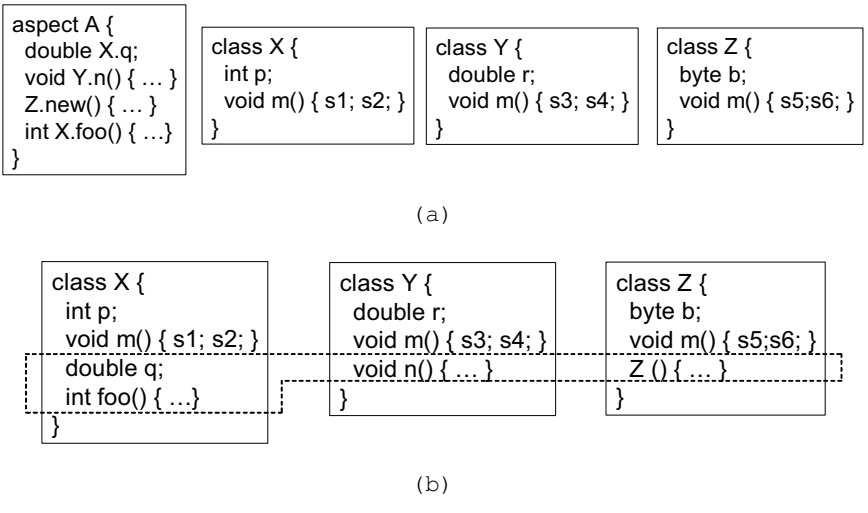

Fig. 1. Static Crosscut Example

[1] AspectJ provides further kinds of static crosscuts which we do not consider for our basic metrics.

The process of applying the crosscutting code to the base code is known as *weaving*. This is performed with an AspectJ compiler such as `ajc` with a command as follows:

```
ajc A.java X.java Y.java Z.java
```

The result of weaving is shown in Figure 1b. Class `X` is augmented with field `q` and method `foo`, class `Y` has a new method `n` and class `Z` has a new constructor. Thus, aspect `A` crosscuts all 3 classes in this example as depicted with a dashed line in Figure 1b[2].

2.2 Dynamic Crosscuts

Dynamic crosscuts run additional code when certain events occur during program execution. The semantics of dynamic crosscuts are commonly described and defined in terms of an event-based model [27][38]. As a program executes, different events fire. These events are called *join points*. Examples of join points are: variable reference, variable assignment, execution of a method body, method call, etc. A *pointcut* is a predicate that selects a set of join points. *Advice* is code executed `before`, `after`, or `around` each join point matched by a pointcut.

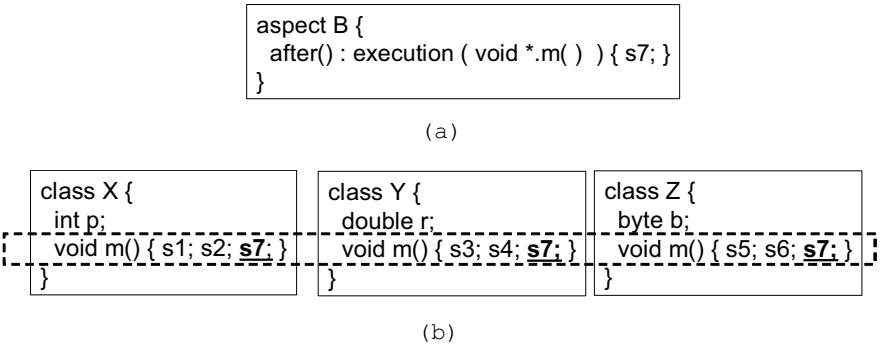

Fig. 2. Dynamic Crosscut Example

Let us consider the example in Figure 2a. Aspect `B` contains a single piece of advice. This advice captures the `execution` of methods `m`, with no arguments and that return `void`, of any type (denoted with `*`). It executes an additional statement, labelled `s7`, `after` the execution of the bodies of method `m`. The result of weaving aspect `B` with classes `X`, `Y`, and `Z` of Figure 1a is shown in Figure 2b, where the additional statement `s7` is added at the end of the method `m` of the 3 classes. Thus in this example the advice in aspect `B` crosscut the 3 classes as depicted with a dashed line in Figure 2b (underlined and bold).

[2] AspectJ generally uses more sophisticated rewrites than those shown in this paper. The composed code snippets we present simplify illustration and are behaviourally equivalent to those produced by `ajc`.

3 Basic Crosscutting Metrics

In this section we provide a semi-formal description of our crosscutting metrics. A goal is to distinguish the contribution to crosscutting stemming from static and dynamic crosscuts. This on one hand sheds light on how the different language constructs are used and on the other it helps to relate aspects with other technologies that can modularize crosscutting concerns.

We describe our using a functional programming style (similar to Haskell [14]) over a simplified abstract program structure. This notation provides a more concise description than natural language and can serve as a guideline for the implementation of tools that automatically gather these and related metrics. We start by describing the abstract structure of our programs, followed by the description of auxiliary functions used to define our metrics.

3.1 Abstract Program Structure

Aspects do not work in isolation. Their functionality is typically implemented in conjunction with a set of classes and interfaces [18][29]. Thus we modularize programs and present our metrics in terms of *features* [39], sets of aspects, classes, and interfaces. Defining our metrics in terms of features permits their application to product line (families of related programs [17]) case studies, an area of increasing interest for the AOP research community [7].

We define a program `P` to be a set of features F_i, denoted with the following list:

$$P=[F_1,F_2,\ldots,F_n]$$

Where `P` is of type `program` and F_i is of type `feature`. Figure 3 summarizes the abstract representation of our programs in the form of a grammar.

A feature `F` consists of a list of feature elements that can be classes, interfaces or aspects. A `class` is a list of `class_element` which can be of type `method`, `constructor`, `field`, etc. An `interface` is a list of `interface_element` which can be of type `field` or method declaration (`methoddecl`). An `aspect` is a list of method (`methodITD`), constructor (`constructorITD`), and field (`fieldITD`) inter-type declarations, and pieces of advice (`advice`). These ITDs are denoted as tuples of class and the corresponding element definition. For example, the tuple for `methodITD` is of type `(class, method)`. For pieces of `advice` we focus only on the pointcut expression `pce` and a `body`. We consider both named and anonymous pointcuts but we focus only on the pointcut expression formed with poincut designators and their combinations denoted with operators `&&`, `||`, `()`, and `!`.

Finally we define an auxiliary type `shadow` with a tuple whose elements are a `program_element` (elements of classes, interfaces and aspects), a `class`, and a pointcut expression `pce`. A *shadow* is a place on the source code whose execution creates join points [32]. We represent a shadow with a tuple of three elements. The first element of `shadow` contains the program element that has the shadow (a `method` for example for a `execution` join point), the `class` that contains the program

element, and pointcut expression `pce` that casts the shadow in that program element. This data structure is not created when programs are originally parsed, instead it is the result of a weaving mechanism.

In this paper we use only the subset of program structures of AspectJ shown in Figure 3. However this abstract program representation can be extended, the same is true for the set of auxiliary functions and metrics we describe in next subsections.

```
program :: [feature]

feature :: [feature_element]
feature_element :: class | interface | aspect

class :: [class_element]
class_element :: method | constructor | ...

interface :: [interface_element]
interface_element :: methoddecl | field

aspect :: [aspect_element]
aspect_element :: methodITD | constructorITD | fieldITD | advice

methodITD :: (class, method)
constructorITD :: (class, constructor)
fieldITD :: (class, field)

advice :: (pce,body)
pce :: pointcut_expression
shadow :: (program_element, class, pce)
program_element :: class_element | interface_element | aspect_element
```

Fig. 3. Abstract Program Representation

3.2 Auxiliary Functions

The following functions provide the basic building blocks of the definitions of our metrics. Note that the names of some of these functions are the plural of the type of element they return as result.

count. This function returns the number of elements in a list. It has signature (where `a` is any type and `n` is a number):

```
count :: [a] -> n
```

loc. This function returns the number of lines of code (LOC). It has signature (where `a` is any type and `n` is a number):

```
loc :: [a] -> n
```

union. N-ary and polymorphic disjoint set union. It receives any number of arguments, unions them and eliminates any repeated elements. We denote its signature with `n` entries of type `b` that when unioned return a list of `b` elements:

$$\texttt{union :: [}b_1\texttt{] -> ...-> [}b_n\texttt{] -> [}b\texttt{]}$$

sum. Receives as input a list of numbers and performs the summation on them. It has the following signature where `n` is a number:

```
sum :: [n] -> n
```

foreach. Receives as input a list and a function. which applies to all the elements in the list. It has signature (where `a` and `b` are any type):

```
foreach :: [a] -> a -> b -> [b]
```

classes. Receives a feature and returns the list of classes in that feature. It has signature:

```
classes :: feature -> [class]
```

interfaces. Receives a feature and returns the list of interfaces in that feature. It has signature:

```
interfaces :: feature -> [interface]
```

aspects. Receives a feature and returns the list of aspects in that feature. It has signature:

```
aspects :: feature -> [aspect]
```

advices. Receives as input a list of aspects and returns the list of pieces of advice contained in the aspects.

```
advices :: [aspect] -> [advice]
```

methodITDs. Receives as input a list of aspects and returns the list of method ITDs or introductions contained in the aspects.

```
methodITDs :: [aspect] -> [methodITD]
```

constructorITDs. Receives as input a list of aspects and returns the list of constructor ITDs or introductions contained in the aspects.

```
constructorITDs::[aspect] -> [constructorITD]
```

ccclasses. This function computes the crosscutting classes from a list of method ITDs, constructor ITDs or field ITDs, and removes any repeated elements. It has signature (where symbol `|` stands for logical `or`):

```
ccclasses :: [ methodITD | constructorITD | fieldITD ] -> [class]
```

pointcuts. Receives as input a list of aspects and returns a list of pointcut designators (`pce`).

```
pointcuts :: [aspect] ->[pce]
```

shadows. This function receives as input a list of pointcuts, finds the join point shadows in a program and returns them in a list:

```
shadows :: [pce] -> [shadow]
```

sclasses. This function receives a list of shadows, extracts their classes (second elements in the shadow tuples), and removes any duplicates.

```
sclasses :: [shadow] -> [class]
```

3.3 Program Structure Metrics

The metrics in this section highlight the contribution of aspects to the overall structure of programs measured in lines of code.

Let `P` be a program. We define the following metrics:

NOF. *Number Of Features.* Counts the number of features in a program.

```
NOF (P) = count (P)
```

NOA. *Number Of Aspects*. Counts the number of aspects in a program.

```
NOA (P) = sum(foreach (P, λf.(count (aspects (f)))))
```

The way to understand this definition is as follows. For each feature `f` in program `P` extract its aspects and `count` them. `Sum` up all the counts of the aspects in all the features.

NCI. *Number of Classes and Interfaces*. Counts the number of classes and interfaces in a program.

```
NCI (P) = sum(foreach (P, λf.(count (union (classes (f)) (interfaces(f))))))
```

BCF. *Base Code Fraction*. Corresponds to the number of lines of code that come from standard Java classes and interfaces relative to the lines of code in a program.

```
BCF(P)= sum(foreach (P, λf.(sum (loc (classes (f)))
                                (loc (interfaces (f))))))
      / loc(P)
```

ACF. *Aspects Code Fraction*. Corresponds to the number of lines of code that come from aspects relative to the lines of code in a program.

```
BCF (P) = sum(foreach (P, λf.(loc (aspects (f))))) / loc(P)
```

IF. *Introductions Fraction*. Corresponds to the number of lines of code that come from introductions or inter-type declarations relative to the lines of code in a program.

```
IF (P) = sum (foreach (P, λf.(sum (loc (fieldITDs (aspects (f))))
                                  (loc (methodITDs(aspects (f))))
                                  (loc (constructorITDs (aspects (f)))))))
       / loc(P)
```

AF. *Advice Fraction*. Corresponds to the number of lines of code that come from pieces of advice relative to the lines of code in a program.

```
AF (P) = sum (foreach (P, λf.(loc (advices (aspects (f)))))) / loc(P)
```

3.4 Feature Crosscutting Metrics

In AOP literature, an *homogenous concern* is one that applies a same piece of advice to several places; whereas an *heterogeneous concern* applies different pieces of advice to different places [7][18]. The metrics in this section adapt these concepts to features and provide a quantitative criteria to classify features within a spectrum that goes from *homogeneous* to *heterogeneous* according to the number and type of crosscuts they implement.

Let `f` be a feature of a program `P`, we define the following metrics:

FCD. *Feature Crosscutting Degree*. Corresponds to the number of classes that are crosscut by all pieces of advice in a feature and those crosscut by the ITDs.

```
FCD(f,P)= count(union(ccclasses(methodITDs(aspects (f))),
                      ccclasses(constructorITDs(aspects (f))),
                      ccclasses(fieldITDs(aspects(f))),
                     sclasses(shadows(pointcuts(advices(aspects(f))),P))))
```

ACD. *Advice Crosscutting Degree*. Corresponds to the number of classes that are crosscut exclusively by the pieces of advice in a feature.

```
ACD(f,P)= count(sclasses(shadows(pointcuts(advices(aspects(f))),P)))
```

HQ. We define *Homogeneity Quotient* as the division of the advice crosscutting degree (ACD) by the feature crosscutting degree (FCD):

```
HQ(f,P) = ACD(f,P)/FCD(f,P) if FCD(f,P)!=0
        = 0                 otherwise
```

PHQ. *Program Homogeneity Quotient*. It corresponds to the summation of the homogeneity quotients for all the features in a program, divided by the number of features NOF.

```
PHQ(P) = sum(foreach(P, λg.HQ(g,P)))/NOF(P)
```

3.5 Homogeneous vs. Heterogeneous Features

We can categorize features according to their Homogeneity Quotient (HQ) within a continuum that has at its ends:

- *Fully Homogenous Feature*: Its pieces of advice crosscut all the classes crosscut by the feature. That is ACD=FCD and thus HQ=1.
- *Fully Heterogeneous Feature*: It is either base code (no crosscutting) or all the crosscutting it does is via ITDs. That is HQ=0.

If the Program Homogeneity Quotient or PHQ tends to value 1 the program is exploiting the crosscutting capabilities of advice. Also, if PHQ tends to value 0, it can have two interpretations: 1) majority class crosscuts are due to inter-type declarations, 2) majority of features have no crosscuts at all. Next section we apply our metrics to four case studies.

4 Case Studies

We applied our set of metrics to four different AspectJ product line systems developed by us and other researchers. They are:

- **ATS.** *AHEAD Tool Suite* is a set of stand alone and language-extensible tools [3] which implement *Feature Oriented Programming (FOP)*, a technology that studies feature modularity in program synthesis for product lines [13]. We performed our study in the AspectJ implementation of five core tools of ATS [30]. Its code is available upon request.
- **FACET.** *Framework for Aspect Composition for an EvenT* channel is an AspectJ implementation of a CORBA event channel, developed at the Washington University [24][3]. The goal of the FACET project is to investigate the development of customizable middleware using AOP. Features in FACET are for example different event types, synchronization, the CORBA core, or tracing.

[3] Source code available at http://www.cs.wustl.edu/~doc/RandD/PCES/facet/

	AHEAD	FACET	Prevayler	AJHotDraw
Program Total Lines Of Code	56727	6364	3964	22104
NOF - Number Of Features	48	34	19	13
NOA - Number Of Aspects	503	113	55	10
NCI - Number of Classes & Interfaces	524	181	107	351
BCF - Base Code Fraction	0.68	0.81	0.69	0.99
ACF - Aspect Code Fraction	0.32	0.19	0.31	0.01
IF - Introductions Fraction	0.19	0.05	0.08	0.01
AF - Advice Fraction	0.01	0.06	0.13	0.00
Other Aspect Code	0.12	0.08	0.10	0.00

Fig. 4. Program Structure Metrics Summary

- **Prevayler.** Prevayler is a Java application that implements transparent persistence for Java objects. In other words, it is a fully-functional main memory database system in which a business object may be persisted. Prevayler was refactored at the Universtiy of Toronto using AspectJ and horizontal decomposition [21][4]. Features are for example persistence, transaction, query, and replication management.
- **AJHotDraw.** AJHotDraw is an aspect-oriented refactoring of the JHotDraw two-dimensional graphics framework. It is an open source software project that provides numerous features for drawing and manipulating graphical and planar objects [1].

4.1 Program Structure Metrics

We applied the first set of metrics to our four case studies. We obtained the following results, summarized in Figure 4:

- **ATS.** The core tools are formed with 48 features for a total 56727 LOC. To the best of our knowledge, we are not aware of any product line in AspectJ of scale comparable to this case study. Base code constitutes 68% of the program code implemented in 524 standard Java classes and interfaces. Aspect corresponds to 32% implemented in 503 aspects. Of this percentage, 19% comes from ITDs, while approximately 1% was contributed by from pieces of advice. The remaining 12% correspond to other constructs such as package imports.
- **FACET.** It consists of 34 features implemented in 6364 LOC. Base code is 81% of total LOC using 181 classes and interfaces. Aspect code is 19% of which 5% are ITDs, 6% are pieces advice, and the remaining 8% comes from other aspect constructs such as aspect methods.
- **Prevayler.** The code base of Prevayler is 3964 LOC modularized into 19 features. Base code is 69% of features LOC and its implemented in 107 classes

[4] Source code available at http://www.msrg.utoronto.ca/code/RefactoredPrevaylerSystem/

and interfaces. The other 31% of total LOC is aspect code, of which 8% comes from ITDs, 13% from pieces of advice, and the remaining 10% from other aspect constructs.

- **AJHotDraw.** It consists of 13 features for a total of 22104 LOC. It is implemented with 351 classes and interfaces and only 10 aspects. Not surprisingly 99% percent of the code is standard Java and only 1% of aspect code, of which almost all comes from ITDs.

4.2 Feature Crosscutting Metrics

ATS. Figure 5 shows the histogram of the homogeneity quotient of the 48 features of ATS. As expected, given the program structure metrics of ATS, most features have no crosscutting, homogeneity quotient of 0. The program homogeneity quotient (PHQ) is 0.03 which indicates a very small use of pieces of advice.

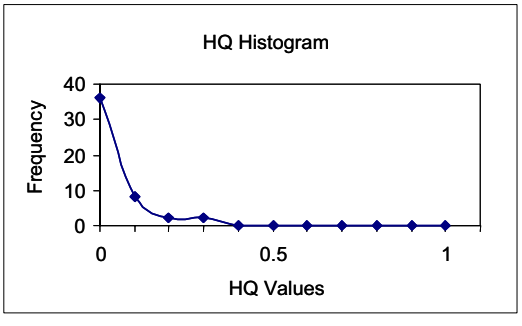

Fig. 5. ATS Homogeneity Quotient Histogram

FACET. Figure 6 shows the homogeneity quotient histogram of the 34 features of FACET. This histogram, as opposed to the one for ATS, has a more balanced distribution, with a program homogeneity quotient whose value is 0.5098.

However, this number has to be put in context. Out of the 34 features of FACET, 22 use pieces of advice. Almost all features that use advice crosscut between 1 and 4 classes, on average 1.3 classes. The exception is a tracing feature that crosscuts all the 181 classes of FACET. Thus, even though around a half of the features are homogeneous the actual impact of advice is limited in terms of the number of classes they crosscut, and the percentage of code they constitute.

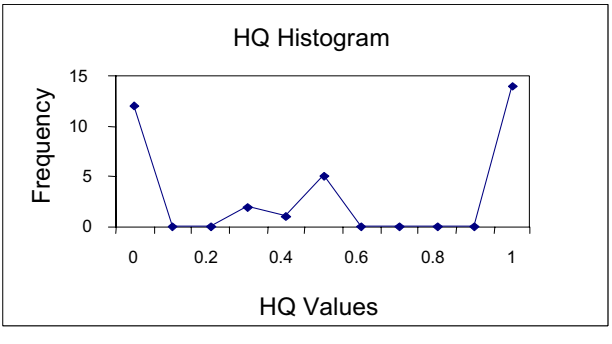

Fig. 6. FACET Homogeneity Quotient Histogram

Prevayler. Figure 7 shows the homogeneity quotient histogram of the 19 features of Prevayler. This histogram shows that most of Prevayler's features are homogeneous with a program homogeneity quotient of 0.7805. Again this result is put in context. On average, each feature crosscuts 3.5 classes, a small percentage of the 107 classes that form Prevayler.

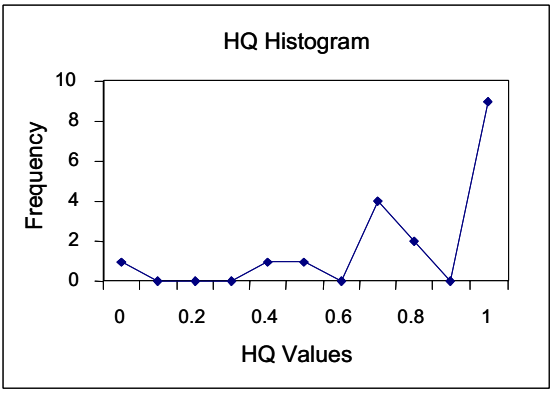

Fig. 7. Prevayler Homogeneity Quotient Histogram

AJHotDraw. Figure 8 shows the homogeneity quotient histogram of the 13 features of AJHotDraw. Given that most of its code is standard Java, its program homogeneity quotient is close to zero 0.0854. Only three of the thirteen features implement crosscuttings: one fully homogenous (uses advice and crosscuts 12 classes), one fully heterogeneous, and one where most crosscutting comes from ITDs (uses advice and crosscuts one class, `HQ` is 0.1111).

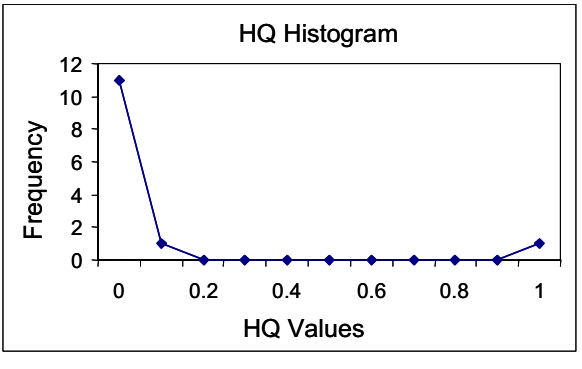

Fig. 8. AJHotDraw Homogeneity Quotient Histogram

5 Collaborations and Heterogeneous Features

Aspects are not the only technique that provides support for crosscutting. There are several techniques categorized as *collaboration-based designs* that also have crosscutting capabilities. This line of research is at least a decade old [23][34][36][37]. A *collaboration* is a set of objects (hence the crosscutting) and a protocol that determines how the objects interact. The part of an object that enforces the protocol in a collaboration is called a *role* [35][37]. One of their goals is to provide a more flexible modularity unit to improve reuse in multiple configurations or compositions for the development of different programs. Thus, collaborations are mechanisms to implement features for product lines [12].

Collaborations can be implemented using several Object Oriented techniques. The kinds of program increments these techniques support are ultimately bound by the Object Oriented ideas they rely upon (i.e. inheritance, polymorphism, encapsulation, etc.). A technique that implements collaborations is FOP and its implementation in AHEAD [13]. For example, using FOP the crosscutting implemented in aspect `A` of Figure 1a is implemented as follows:

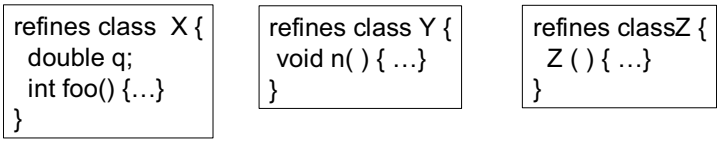

Fig. 9. Crosscutting Example in FOP

The kinds of crosscutting that AHEAD and other collaboration-based designs techniques support correspond to AspectJ's static crosscutting inter-type declarations that we considered. In other words, the distinctive characteristic of aspects is its support for dynamic crosscuts implemented with pieces of advice.

We have seen that in the four case studies we analysed, the pieces of advice crosscut a relatively small number of classes in comparison with the number of classes in the entire programs. Furthermore, the percentage of lines of code is also small, ranging from 1% to 13% in our examples, on average 6%. These numbers beg the questions: Are these results typical? What is the real impact of aspects in software development if their distinctive trait is advice?

We claim that these results are not atypical. Our experiences and other's working with product lines and aspect programs lead us to conjecture that most of the features or crosscuttings in programs are of heterogeneous nature, and that pieces of advice crosscut few classes relative to the number of classes that build a system [8][30][18]. Intuitively, the reason behind this conjecture is that large programs are not synthesized by adding the same piece of code in different places, but rather, adding different pieces of code in different places [6].

Our response to the second question is that aspects can be extremely useful for modularizing crosscutting that involves many classes such as logging, however these types of crosscutting are not pervasive in all software systems and constitute a small fraction of the overall code.

6 Related Work

Several metrics have been proposed for aspects. Zhao and Xu describe metrics for aspect cohesion based on aspect dependencies graphs [41]. Zhao also utilizes a similar framework to define measurements for aspect coupling [40]. Their metrics are formally described, however they lack concrete architectural interpretation and, to the best of our knowledge, have not been applied to actual case studies.

Coupling metrics have been proposed by Ceccato and Tonella [15]. They extend and adapt to AOP some of Chidamber and Kemerer's metrics for Object Oriented systems [16]. This set of metrics is defined informally and it is applied to a tiny case study (250+ LOC), Hannemann's implementation of the Observer Pattern [22]. However, its is unclear how these metrics would extrapolate to larger case studies and their architectural significance.

Bartsch and Harrison evaluate five metrics in Ceccato and Tonella's work [11]. They argue that only one of the evaluated metrics can be considered well-defined (lacks any interpretation ambiguities), and none of them are completely valid from a measurement theory point of view. Along the same lines, Mehner proposes a series of steps to validate AOP metrics and their application [33].

An extensive study on modularizing design patterns have been performed by Garcia et al. [20]. They use Hannemann's implementation of GoF patterns to apply seven metrics that extend and adapt to AOP Chidamber and Kemerer's metrics [16]. Their metrics are informally defined and their results are given an interpretation in terms of improvement of separation of concerns and reuse.

Coupling metrics for AOP certainly depend on the crosscutting capabilities of aspects. Our metrics focus only on crosscutting relations produced by pointcut shadows and ITD's, and do not consider cases such as method calls or field references which the above coupling metrics account for.

7 Conclusions and Future Work

In this paper we present a semi-formal description of a set of crosscutting metrics. Our metrics categorize crosscutting within an spectrum from heterogeneous to homogeneous depending on the number of classes crosscut by pieces of advice in relation to the number of classes crosscut by all crosscutting mechanisms of AspectJ. This distinction helps draw a relation with other technologies that also provide support for crosscutting.

We applied our set of metrics to four case studies. We found that for these programs, the number of classes crosscut by advice per crosscutting is small in relation to the number of classes in the program, and that crosscuttings are mostly heterogeneous. We argued that this finding is not atypical as programs are not synthesized by adding the same piece of code in different places, but rather, adding different pieces of code in different places. We are in the process of applying our metrics to other case studies to provide more empirical arguments to further support our conjecture.

Earlier work of the first author described a preliminary definition of our metrics that were applied to a single case study [31]. Work of the second author categorizes

crosscuts in two dimensions [7][9]. We plan to integrate these two dimensions into the set of metrics presented here. We also intend to extend our metrics to address issues such as cohesion and coupling for features. These extended metrics could help identify opportunities for feature refactoring.

We collected the Program Structure Metrics with the *AJStats* tool. This tools is under development at the University of Magdeburg and it is publicly available [2]. Currently we are collecting the Feature Crosscutting Metrics manually. We are exploring different possibilities to extend AJStats to collect this set of metrics. Our goal is to develop tool infrastructure that would allow the implementation of these and other metrics in a simple and extensible way.

References

1. AJHotDraw project web site http://sourceforge.net/projects/ajhotdraw .
2. AJStats tool project website http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/ajstats/ .
3. AHEAD Tool Suite (ATS). http://www.cs.utexas.edu/users/schwartz
4. Alves, V., Matos, P., Cole, L., Borba, P., Ramalho, G.: Extracting and Evolving Game Product Lines. SPLC (2005)
5. Anastasopoulus, M., Muthig, D.: An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology. ICSR (2004)
6. Apel, S.: The Role of Features and Aspects in Software Development. PhD Dissertation. School of Computer Science, University of Magdeburg, 2007.
7. Apel, S., Leich,T., Saake. G.: Aspectual Mixin Layers: Aspects and Features in Concert. ICSE (2006)
8. Apel, S., Batory, D.: When to Use Features and Aspects? A Case Study. GPCE (2006)
9. Apel, S., Batory, D.: On the Structure of Crosscutting Concerns: Using Aspects or Collaborations?. AOPLE (2006)
10. AspectJ, *http://eclipse.org/aspectj/.*
11. Bartsch, M., Harrison, R.: An Evaluation of Coupling Measures for AspectJ. LATE Workshop AOSD (2006)
12. Batory, D., Cardone, R., Smaragdakis, Y.: Object-Oriented Frameworks and Product-Lines. SPLC (2000)
13. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE TSE, June (2004)
14. R. Bird.: Introduction to Functional Programming using Haskell. Prentice Hall (1998)
15. Ceccato, M., Tonella, P.: Measuring the Effects of Software Aspectization. First Workshop on Aspect Reverse Engineering. Delft, The Netherlands (2004)
16. Chidamber, S., Kemerer, C.: A Metrics Suite for OOD Design. IEEE TSE 20(6) (1994)
17. Clements, P., Northrop, L.: Software product lines: practices and patterns. Addison-Wesley (2002)
18. Coyler, A., Clement, A.: Large-scale AOSD for Middleware. AOSD (2004)
19. Coyler, A., Rashid, A., Blair, G.: On the Separation of Concerns in Program Families. TRCOMP-001-2004, Computing Department, Lancaster University, UK (2004)
20. Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., von Staa, A.: Modularizing Design Patterns with Aspects: A Quantitative Study. Transactions on TAOSD I. LNCS 3880 (2006)
21. Godil, I., Jacobsen, H.-A.: Horizontal Decomposition of Prevayler. CASCON (2005)

22. Hannemann, J.: AspectJ implementation of GoF patterns. http://www.cs.ubc.ca/~jan/AODPs
23. Holland, I.: Specifying Reusable Components using Contracts. ECOOP (1992)
24. Hunleth, F., Cytron, R.: Footprint and Feature Management Using Aspect-Oriented Programming Techniques. In Proceedings of the Joint Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES/SCOPES), pages 38~V45 (2002)
25. Kiczales, G., Hilsdale, E., Hugunin, J., Kirsten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. ECOOP (2001)
26. Laddad, R.:. AspectJ in Action. Practical Aspect-Oriented Programming. Manning (2003)
27. Lämmel, R.: Declarative Aspect-Oriented Programming. PEPM (1999)
28. Lopez-Herrejon, R.E., Batory, D., Cook, W.: Evaluating Support for Features in Advanced Modularization Techniques. ECOOP (2005)
29. Lopez-Herrejon, R.E., Batory, D., Lengauer, C.: A disciplined approach to aspect composition. PEPM (2006)
30. Lopez-Herrejon, R.E., Batory, D.: From Crosscutting Concerns to Product Lines: A Function Composition Approach. Tech. Report UT Austin CS TR-06-24. May (2006)
31. Lopez-Herrejon. R.E.: Towards Crosscutting Metrics for Aspect-Based Features. AOPLE Workshop at GPCE (2006)
32. Masuhara, H., Kiczales, G.: Modeling Crosscutting Aspect-Oriented Mechanisms. ECOOP (2003)
33. Mehner, K.: On Using Metrics in the Evaluation of Aspect-Oriented Programs and Designs. LATE Workshop AOSD (2006)
34. Reenskaug, T., Anderson, E., Berre, A., Hurlen, A., Landmanrk, A., Lehne, O., Nordhagen, E., Ness-Ulseth, E., Ofdetal, G., Skaar, A., Stenslet, P.: OORASS : Seamsless Support for the Creation and Maintenance of Object-Oriented Systems. Journal of Object Oriented Programming, 5(6): (1992)
35. Smaragdakis, Y., Batory, D.: Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. ACM TOSEM April (2002)
36. Van Hilst, M., Notkin, D.: Using C++ Templates to Implement Role-Based Designs. JSSST International Symposium on Object Technologies for Advanced Software. Springer-Verlag (1996)
37. Van Hilst, M., Notkin, D.: Using Role Components to Implement Collaboration-Based Designs. OOPSLA (1996)
38. Wand, M., Kiczales, G., Dutchyn, C.: A Semantics for Advice and Dynamic Join Points in Aspect Oriented Programming. TOPLAS (2004)
39. Zave, P.: FAQ Sheet on Feature Interaction. http:// www.research.att.com/~pamela/ faq.html
40. Zhao, J.: Measuring Coupling in Aspect-Oriented Systems. Technical Report SE-142-6. Information Processing Society of Japan (IPSJ), June (2003)
41. Zhao, J., Xu, B.: Measuring Aspect Cohesion. FASE (2004)