Abstract

While it is well known that crosscutting concerns occur in many software projects, little is known about the inherent properties of these concerns nor how aspects (should) deal with them. We present a framework for classifying the structural properties of crosscutting concerns into (1) those that benefit from AOP and (2) those that should be implemented by OOP mechanisms. Further, we propose a set of code metrics to perform this classification. Applying them to a case study is a first step toward revealing the current practice of AOP.

2. Crosscut Classification Framework

2.1 Homogeneous and Heterogeneous Crosscuts

A homogeneous crosscut extends a program at multiple join points by adding one extension, which is a coherent piece of code [10]. For example, an advice may advise a whole set of method executions, i.e., it adds new classes and interfaces as well as injects new fields, methods, interfaces, and super-classes etc. Inter-type declarations are examples of static crosscuts (first row of Table 1).

A heterogeneous crosscut extends multiple join points by adding multiple extensions, where each individual extension is implemented by a distinct piece of code that affects exactly one join point [10]. For example, an aspect might bundle a set of advice that extends a set of methods, whereby each advice extends exactly one method; or an aspect bundles a set of inter-type declarations – each intended for a distinct class (right column of Table 1).

2.2 Static and Dynamic Crosscuts

A static crosscut extends the structure of a program statically [29], i.e., it adds new classes and interfaces as well as injects new fields, methods, interfaces, and super-classes etc. Dynamic crosscuts are especially interesting when they exceed the level of known events such as method calls or executions. Work on AOP suggests that expressing a program extension in terms of sophisticated events increases the abstraction level and captures the programmer’s intention more directly. There are proposals for new language constructs for defining and catching new kinds of events during the program execution [26,30].

Basic and advanced dynamic crosscuts. Dynamic crosscuts are handled by aspects (should) deal with them. We present a framework for classifying the structural properties of crosscutting concerns into (1) those that benefit from AOP and (2) those that should be implemented by OOP mechanisms. Further, we propose a set of code metrics to perform this classification. Applying them to a case study is a first step toward revealing the current practice of AOP.

We propose four metrics to analyze aspect-oriented programs to make the above distinctions, i.e., do the aspects of a program implement crosscutting concerns that really demand AOP language mechanisms? We are building a tool that will collect data from a representative spectrum of software projects that employ AOP. We discuss the data for one AspectJ project exemplarily.

1. Introduction

While many studies have examined the capabilities of aspect-oriented programming (AOP) to improve the modularity, customization, and evolution of software [8, 9, 13, 14, 21, 38], little is known on how AOP has been used. We are interested in knowing which language mechanisms are used in current aspect-oriented programs, to what extent, and for what kinds of problems. Knowing this helps (1) build AOP tools that reflect the programmer’s needs; (2) provide programming guidelines for exploiting AOP mechanisms better, i.e., it adds new classes and interfaces as well as injects new fields, methods, interfaces, and super-classes etc. Inter-type declarations are examples of static crosscuts (first row of Table 1).

To address these issues we propose a framework for classifying crosscutting concerns (a.k.a. crosscuts). Our framework enables us to assign individual crosscuts to two distinct categories: (1) crosscuts that really demand AOP mechanisms and (2) crosscuts that can be implemented appropriately using well-known OOP mechanisms. This distinction follows a long line of prior work on collaboration-based designs [31,32,35], feature-oriented programming [4], and design patterns [12]. All of them advocate object-oriented mechanisms for a certain class of design and implementation problems, so called collaborations, which fall into one category.

We propose four metrics to analyze aspect-oriented programs to make the above distinctions, i.e., do the aspects of a program implement crosscutting concerns that really demand AOP language mechanisms? We are building a tool that will collect data from a representative spectrum of software projects that employ AOP. We discuss the data for one AspectJ project exemplarily.

1http://www.eclipse.org/aspectj/
3. Two Categories of Crosscutting Concerns

We argue it is crucial to decide which crosscutting concerns should be implemented as aspects, and how, and which using traditional object-oriented techniques. For that purpose, we divide the space of possible crosscuts that is defined by our classification framework into two categories, (1) those crosscuts that abstract collaborations and (2) those that address the dynamic program semantics and/or that are homogeneous. The two categories map roughly to the two programming paradigms, OOP and AOP.

3.1 Collaborations

A collaboration of classes is a set of classes that communicate with one another to implement a semantically coherent piece of functionality. Classes of a program play different roles in different collaborations [35]. A set of collaborating classes being added to a program can be understood as a feature of that program [4]. That is, a collaboration extends a program by adding new classes and by applying new roles to existing classes, whereby each role is implemented as a refinement (e.g., using virtual classes [25] or mixins [6]). From that perspective, a role adds new elements to a class and extends existing elements, such as methods.

Figure 1 depicts a collaboration-based design of a graph implementation, where the classes Graph, Node, and Edge collaborate together. The feature WeightedGraph adds the class Weight and extends the classes Graph and Edge simultaneously. For example, the class Edge plays two roles, one in the BasicGraph and one in the WeightedGraph.

A significant body of work has observed that collaborations of classes are predominantly of a heterogeneous structure [4, 5, 20, 29, 32–35]. That is, the roles and classes added to a programs differ in their functionality, as in our graph example. Hence, a collaboration is a heterogeneous crosscut and a heterogeneous crosscut can be understood as collaboration applied to a program. Therefore, it is straightforward to employ from techniques for encapsulating that is evaluated at runtime, e.g., a method execution is only advised if it occurs in the control flow of another method execution.

3. Basic dynamic crosscuts address events known from OOP; advanced dynamic crosscuts can specify composite events that trigger the execution of an extension, e.g., trace matches are executed when events fire in a specific pattern thereby involving the history of computation [1].

With AOP, an advanced dynamic crosscut is implemented by an advanced advice and a basic dynamic crosscut by a basic advice. The distinction between basic and advanced advice is useful to identify which pieces of advice make use of advanced AOP mechanisms and which pieces of advice mimic well-known OOP method extensions.

<table>
<thead>
<tr>
<th>homogeneous</th>
<th>heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>void Point.setX(int x)</td>
</tr>
<tr>
<td></td>
<td>{ /* * */ }</td>
</tr>
<tr>
<td>basic</td>
<td>before() : execution(void Point.setX(int))</td>
</tr>
<tr>
<td>dynamic</td>
<td>{ /* * */ }</td>
</tr>
<tr>
<td>advanced</td>
<td>before() : execution(void Point.setX(int))</td>
</tr>
<tr>
<td>dynamic</td>
<td>{ /* * */ }</td>
</tr>
</tbody>
</table>

Table 1. A classification framework for crosscutting concerns (AspectJ examples).

and composing object-oriented collaborations when implementing heterogeneous crosscuts [6, 25, 29, 32, 35].

3.2 Homogeneous and Advanced Dynamic Crosscuts

Crosscuts that do not fall in the above category are either homogeneous crosscuts and/or advanced dynamic crosscuts.

Aspects perform well in extending a set of join points using one coherent advice or one localized inter-type declaration, thus, modularizing a homogeneous crosscut. Thereby, programmers avoid code replication. Figure 2 depicts an aspect that implements the feature Color, which is homogeneous. It defines an interface for colored entities (Line 2) and declares via inter-type declaration that Node and Edge implement that interface (Line 3). Furthermore, it introduces via inter-type declarations a field color (Line 4) and two accessor methods to Node and Edge (Lines 5-7,8-10). Finally, it advises the execution of the method print of all colored entities to change the display color (Lines 11-13).

3.3 WeightedGraph

WeightedGraph extends the classes Graph and Edge and implements the interface Weight (Line 4). It declares two new fields weight and color of type Color (Lines 5-6) and introduces two accessor methods to Node and Edge (Lines 8-9). This feature advises the execution of the method print of all colored entities to change the display color (Lines 11-13).

Advice is well-suited for implementing advanced dynamic crosscuts [29]. When advising the printing mechanism of our graph implementation we can take advantage of the sophisticated mechanisms of AOP. Background is that the print methods of the parent...
ticipants of the graph implementation call each other (especially, composite nodes that call print of their inner nodes). To make sure that we do not advise all calls to print, but only the top-level calls, i.e., calls that do not occur in the dynamic control flow of other executions of print, we can use the cflowbelow pointcut as conditional (Fig. 3). This is an example of an advanced advice.

```java
aspect PrintHeader {
  before() : execution(void print()) &&
  cflowbelow(execution(void print())) { header(); }
  void header() { System.out.println("header:␣"); }
}
```

Figure 3. Advising print advanced advice.

Though language abstractions such as cflow and cflowbelow can be implemented (emulated) using traditional OOP, usually that results in code replication, tangling, and scattering.

3.3 Discussion

Table 2 depicts the guidelines for using AOP and OOP mechanisms based on their individual strengths. First, aspects should be used for modularizing homogeneous crosscuts to avoid code replication. Second, aspects avoid code scattering and tangling in case of using advanced advice for implementing advanced dynamic crosscuts. For heterogeneous crosscuts which extend only methods and classes, OOP techniques for collaboration-based designs suffice. It has been observed that although both approaches are able to implement the crosscuts of the other, they cannot do so elegantly [2,3,29].

<table>
<thead>
<tr>
<th>homogeneous</th>
<th>heterogeneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>set of roles that add elements</td>
</tr>
<tr>
<td>basic</td>
<td>set of roles that override methods</td>
</tr>
<tr>
<td>dynamic</td>
<td>set of advanced advice</td>
</tr>
</tbody>
</table>

Table 2. What implementation technique for what kind of crosscutting concern?

4. Metrics

We propose a set of metrics to provide insight into the current practice of AOP. They enable to decide in which category a given aspect falls. The metrics are quantified by the number of occurrences (NOO) of a certain software artifact and/or the lines of code (LOC) associated with it.

**Classes, interfaces, and aspects (CIA).** The CIA metric determines the NOO of classes, interfaces, and aspects, as well as the LOC associated with each. It tells us if aspects (as opposed to classes and interfaces) are a small or a large fraction of the used modularization mechanisms in a software project, and if these implement a significant or only a small part of the code base of that project.

**Homogeneous crosscuts (HC).** The HC metric measures the extent in which homogeneous and heterogeneous crosscuts are used. We calculate the fraction of advice and inter-type declarations that implement homogeneous crosscuts (NOO) and the fraction of the code base that is associated with them (LOC). The HC metric tells us if the aspects of a program exploit the pattern-matching mechanisms of AOP or merely emulate OOP mechanisms.

**Advanced dynamic crosscuts (ADC).** This metric determines the NOO of advanced advice and the overall LOC associated with them. It tells us to what extent aspects make use of the advanced capabilities of AOP for implementing dynamic crosscuts.

**Code replication reduction (CRR).** The CRR metric determines the reduction in LOC when using homogeneous advice and inter-type declarations, as opposed to the LOC resulting from using traditional OOP mechanisms. The code reduction for one piece of homogeneous advice / inter-type declaration is roughly the number of affected join points, multiplied by the LOC associated with them.

5. Collecting Statistics of AspectJ Programs

**CIA metric.** Collecting data for the CIA metric we traverse all source files of a given project and calculate the number and LOC of aspects, classes, and interfaces – excluding blank lines and comments.

**HC metric.** Homogeneous crosscuts are indicated by inter-type declarations and advice that contain wildcards (* and +). If we discover logical operators in pointcuts that combine two pointcuts of the same type (e.g., execution(...) || execution(...)) then the advised advice are also counted as homogeneous. Inter-type declarations that contain logical operators are considered homogeneous as well as advice that do not qualify a target method or field completely, e.g., by omitting the type or the arguments.

**ADC metric.** We define all advice as advanced advice except those associated to call and execution that are not combined with any other pointcuts, except with target and args (execution can also be combined with this). This is an overestimation that might consider some pieces of advice that are not advanced as advanced advice, but not vice versa. The remaining advice are considered basic advice.

**CRR metric.** For determining the code reduction due to eliminating replicated code, we determine the number of join points per homogeneous advice and inter-type declaration. We multiply the number of join points minus one for each advice or inter-type declaration, with the LOC associated. Finally, we sum up the saved LOC of all advice and inter-type declarations to get the overall code reduction.

6. A Case Study

As case study we analyzed FACET (6364 LOC), an AspeCJ-based CORBA event channel, implemented at the Washington University [18]. We used our tool AJStats for collecting the NOO and LOC of all artifacts of FACET. We determined the properties of advice / inter-type declarations and the caused code reduction by hand.

Table 3 depicts our collected statistics. Column NOO lists the number of artifacts we found of a specific type (e.g., homogeneous advice) and its fraction with regard to the overall number of this type (e.g., all pieces of advice). Column LOC depicts the LOC associated with a certain kind of artifact and its fraction of the overall code base. In the following paragraphs we examine the data in depth.

5Recall that advanced advice can be either heterogeneous or homogeneous (cf. Fig. 1).

6Although the semantics of call is to advise the client side invocations of a method, it can be implemented as method extension – preconditioned that all calls to the target method are advised; the above definition ensures that.

7http://wwwitis.cs.uni-magdeburg.de/iti/db/ajstats/
CIA metric. FACET uses relatively many aspects, compared to other studies [2, 8, 9, 21, 38]. This observation is remarkable since it demonstrates that aspects are used in different software projects to a different extent. 38% of all modularization mechanisms were aspects, which occupied 19% of the overall code base.

HC metric. In FACET we found 4 of 49 pieces of advice and 8 of 113 inter-type declarations were homogeneous. That is, 7% of all implemented crosscuts were homogeneous, which occupied 0.4% of the overall code base. In contrast, 93% of all crosscuts were heterogeneous, occupying 9% of the code base.

ADC metric. We found 11 of 49 advice were advanced advice. They are associated to cflow pointcuts or use the returning clause. That is, 22% of all advice were advanced advice, which occupied 2% of the overall code base. The remaining 38 advice were basic advice, which occupied 3% of the overall code base.

CRR metric. 4 pieces of advice and 8 inter-type declarations are homogeneous. We calculated the effective code reduction of 534 LOC, which is a 8% reduction compared to a version that uses OOP mechanisms for implementing homogeneous crosscuts.

Acknowledgments
Sven Apel is sponsored by the German Research Foundation (DFG), project number SA 465/31-1 and SA 465/32-1. Marko Rosenmüller is sponsored by the German Research Foundation (DFG), project number SA 465/32-1. The presented study was conducted when Sven Apel was visiting the group of Don Batory at the University of Texas at Austin. Batory’s research is sponsored by NSF’s Science of Design Project #CCF-0438786.

References