Aspects developed in aspect-oriented systems often need to hook onto multiple objects that share common structural characteristics - such as attributes and operations. In strongly-typed aspect-oriented systems like AspectJ these objects need to be of common type so that pointcuts may designate them and pieces of advice may interact with them. Such type-systems are typically based on nominal types, therefore, aspects cannot interact with objects according to their structural information in a common way. This paper argues that specifying aspects based on a nominal type system is not sufficient and shows that aspect-specifications based on structural characteristics overcome this problem. A corresponding extension of the nominal type systems is proposed and illustrated by means of structural types and compound types.

1. INTRODUCTION

In aspect-oriented software development [12], aspect-oriented systems such as AspectJ [13, 16] permit to weave aspects to join points within the base application according to underlying selection criteria specified by using a pointcut language (cf. for example [6, 9]). Additionally, aspect-oriented systems permit to adapt such join points using mechanisms like e.g. advice. Furthermore, aspect-oriented systems typically need to interact with the objects that participate in a certain join point. Consequently, aspects interact with the base application not only by capturing messages and changing a message’s parameters but also by explicitly invoking methods of such objects. For this purpose, systems like AspectJ provide a mechanism called context exposure (cf. e.g. [8]) to pass an object from a join point’s context to an advice’s context.

AspectJ, heavily relies on Java’s type system. Objects within an advice have a type, which is used for type checking. The compiler checks whether it is valid to send a certain message to a type, or it checks whether a certain object (having a certain type) can be used as a parameter for a certain message. The type system of Java is based on conformance by name [15], i.e. types and type relationships are determined by a type’s name.

It turns out that such a type system is sometimes inappropriate for aspect-oriented software development. There are a number of situations where only structural information about the types being involved in a certain join point is known when the aspect is specified – but structural information is not sufficient to specify a corresponding nominal type. Consequently, it is not easy to specify a corresponding advice if the aspect needs to interact with such objects.

In this paper, we motivate the need for utilizing structural information for the specification of types in an aspect-oriented system and introduce an extension of the language AspectJ that utilizes structural types [15] and compound types [3], i.e. the ability to use structural type information.

In section 2 we motivate the need for types in AspectJ based on structural information and discuss the reasons why this problem occurs especially in aspect-oriented systems. In section 3 we introduce an AspectJ extension that includes structural and compound types. After discussing some related work in section 4, we discuss and conclude this paper in section 5.

2. MOTIVATION

2.1 Example 1: Closeable Objects

Within a multi-user application users are permitted to open a number of connections to different kinds of resources such as database connections and file streams connections. Hence, it is desirable to close connections if they are no longer needed. The point in time (i.e. the join point) when such resources are no longer needed can be clearly identified: At least when the current user logs out, all connections can be closed.

```
aspect CloseAllCloseableObjects {
    after(): execution(*.logout(..)) {
        Iterator it = getAllOpenConnections().iterator();
        while(it.hasNext()) { // some exception handling...
            ((CloseAble) it.next()).close();
        }
        Collection getAllOpenConnections() { // ... }
    }
}
```

Figure 1. Aspect for closing all closeable objects.

From an aspect-oriented perspective it is desirable to handle the release of resources in one module. In Figure 1, the CloseAllCloseableObjects aspect collects all objects that have to be closed when the user logs out and closes them. The method openConnections() returns a collection that contains all objects representing open connections. The advice iterates over all objects, casts them to a type CloseAble, and sends them the message close.

However, the problem is that the method close is spread over a number of types that do not share a common supertype. Hence, it is not possible to cast them to a common type. Figure 2 illustrates a number of types whose objects should potentially be handled by the aspect. All types have in common that they provide a method close that does not have a return type. However, the methods differ in respect to their exceptions: Some of them do not have any
2.2 Example 2: Mouse Listeners For Applets

In AspectJ, it is possible to provide a common interface to different types by using introductions; declare parents in combination with implements permits to add a new interface to a number of classes as applied in in Figure 3. As a consequence, the aspect CloseAllCloseableObjects introduces the new interface CloseAble to all target types. The target types are enumerated and the new interface is added to them.

```java
interface CloseAble { void close() throws Exception; }

aspect CloseAllCloseableObjects {
    declare parents:
        (javax.sql.Connection || javax.sql.ResultSet ||
         java.io.Reader || java.io.OutputStream ||
         java.net.Socket || java.nio.channels.Channel ||
         javax.sound.midi.MidiDevice .../* additional types */) 
    implements CloseAble;
    ...
}
```

Figure 3. Introducing CloseAble to target classes.

However, there are a number of reasons why this approach is not desirable. First, the developer needs to find and enumerate all types structurally matching CloseAble. This is too time consuming in large-scale applications. Second, too easily developers may simply forget to add a certain type. Consequently, the aspect does not perform the desired functionality. Third, whenever the underlying system evolves in a way that new types are added to the application that are also closeable, the developer is enforced to extend the enumeration of types.

### 2.3 Problem Statement

The overall problem in the previously illustrated examples is that aspect-specific behavior needs to be executed for different types which share common structural elements but which do not have a parameter of type MouseListener, respectively MouseMotionListener. In such approaches, the applet needs to send these messages to itself with the parameter this as illustrated in Figure 4. Since the behavior of adding an applet implementing both interfaces occurs over and over again, it seems desirable to factor out this behavior to be handled by an aspect.

```java
public class MyApplet extends Applet {
    implements MouseListener, MouseMotionListener {
        // some instance variables
        ...
        public void mousePressed(MouseEvent e){...} ...
        public void init() {
            addMouseListener(this);
            addMouseMotionListener(this);
        } ...
    }
}
```

Figure 4. Recurring applet implementations using MouseListener and MouseMotionListener.

Similar to the first example, this can be achieved by defining a new marker interface (ListeningComponent in Figure 5). The developer has to introduce the corresponding interface to all target types that fulfill the interface. However, the quality of the problem here is different.

```java
interface ListeningComponent {
    extends MouseListener, MouseMotionListener {
        public void addMouseListener(MouseListener l);
        public void addMouseMotionListener(MouseMotionListener l);
    }

    aspect RegisterMouseListener {
        after(ListeningComponent c) : c.addMouseMotionListener(..)
        && this[c] { c.addMouseListener(c); c.addMouseMotionListener(c); }
    }
}
```

Figure 5. Listener registration aspect.

First, in this example, the aspect explicitly refers to classes that do not exist in the Java SDK: the classes are either project-specific classes, or classes that might be developed in the future. As a consequence, it is not possible for an aspect to introduce the corresponding interface already to a number of classes since they are not known when the aspect is specified. Second, the target type is not identified purely because of its methods, but because of the implemented interfaces. However, the knowledge that the target classes need to implement these two interfaces cannot be utilized: By using introduction it is only checked whether the target class is able to fulfill the corresponding interface.

In case developers would be able to refer directly to all types that implement MouseListener and MouseMotionListener and extend Applet: the above problems would directly vanish.

### 2.3 Problem Statement

Note that developers can also use anonymous classes with the corresponding Adapter-classes, but further discussion of design alternatives for the problem is out of the scope of this paper.

1. Note that the declared exception in method close() is the supertype of any other declared exception, moreover, some implementing classes don’t declare an exception. According to the Java language specification [11] this is valid.

2. Note that developers can also use anonymous classes with the corresponding Adapter-classes, but further discussion of design alternatives for the problem is out of the scope of this paper.
corresponding (nominal) type in common (except Object). Aspects
on the other hand need to handle instances of these types in a
unique way, i.e. there is an aspect-specific type that needs to
reflect the structural commonalities of different types. Such aspect-
specific type serves three purposes. First, they permit to specify
where aspect-specific code needs to be executed. Second, they are
used within an advice’s body to define a common behavior at the
concerning join points. Third, they are used to exposure typed
objects from the pointcut to the advice.

However, base applications that aspects reason about do not
provide a common type for different classes. Types from different
base applications are developed independent from each other in
different projects. This corresponds to the problem of combining
independently emerged frameworks (see for example [6, 18]):
frameworks that are developed independent of each other.
However, even in independent developed types there are structural
similarities that result from common naming conventions, guidelines,
idioms or commonalities between domain models. Although there is
some structural knowledge available about the types, such
knowledge is not sufficient in order to specify the corresponding
aspects because type information is missing.

The origin of the problem comes from the underlying type system:
In Java (and AspectJ) types are identified by their names (see for
element [9, pp. 251-253] for a discussion between nominal and
structural types.). For example, a type B is a subtype of another
type A, if B directly or indirectly extends (or implements) A. I.e. the
relationship between two types depends on how corresponding
relationships are declared in interfaces or classes. Knowledge about
certain ingredients of types are not sufficient to specify a
corresponding type: Either a type’s name is known and can be used
for typing or the type’s name is unknown and there is no possibility
to specify a corresponding type. However, to specify aspects from
the motivating examples, it is necessary to refer to objects because
of structural properties.

As illustrated above, AspectJ’s current solutions suffer from the
problem of enumeration-based crosscutting [6]. Consequently, a
mechanism is needed in AspectJ to specify aspects that refer to the
base application by using structural type information.

3. STRUCTURAL TYPES AND COMPOUND TYPES FOR ASPECT-SPECIFICATIONS

To overcome the previously discussed problems we see the need to
add structural types [15] to AspectJ. I.e. there are types that are
implicitly implemented by all types having corresponding signatures.
However, as we will see, only structural types are not sufficient to
overcome the mentioned problems. Therefore, we also adapted the
idea of compound types [3] which permits to utilize sets of types to
determine whether or not a certain type matches.

3.1 Structural Types

In order to add structural typing to AspectJ a special type construct
structural is required that declares structural types. A
structural declaration corresponds to an interface declaration, i.e. its
body declares methods. The subtype relationships between nominal
types and structural types are computed in a different way: Nominal
types do not explicitly define a subtype relationship to structural
types. Instead, all types that structurally match a structural type are
subtypes of the structural type.

Figure 6 illustrates an example of a structural type according to the
example from section 2.1. A structural type Closeable is
declared. Hence, all types that structurally match Closeable are
subtypes of it. Since the classes Connection and Reader provide
a method whose signature is equal to the method close in
Closeable, Connection and Reader are implicitly subtypes of
Closeable (the implicit subtype relationship is illustrated by an
UML-implements relationship in Figure 6).

As illustrated above, AspectJ’s current solutions suffer from the
problem of enumeration-based crosscutting [6]. Consequently, a
mechanism is needed in AspectJ to specify aspects that refer to the
base application by using structural type information.

3.2 Compound Types

In the second example the developer wants to invoke
addMouseListener and addMouseMotionList from
within the advice on the classes extending Applet and that
implement MouseListener and MouseMotionListener with
the same object as a parameter. Since the aspect’s advice just needs to
invoke both methods on a certain object, the problem could be
reduced in principle to some extent to a structural type that defines
"a type that implements all methods declared in MouseListener

Figure 6. Structural type Closeable.
and in MouseMotionListener, and that has 
addMouseListener and addMouseMotionListener methods”. While we keep the information about the implemented interfaces, we lose information about “extends Applet” and reduce it to the expected methods. This loss of information potentially leads to an accidental matching: The aspect would also address objects that do not extend Applet. Furthermore, this loss of information makes it harder to understand the intention of the specified structural type and the pointcut specifications.

To overcome this problem, we see the need to specify not only signatures in structural types that are compared with signatures within nominal types. We also want to specify elements from the inheritance hierarchy of the addressed types.

Compound types as introduced in [3] for the programming language Java represent a solution to this problem. A compound type consists of an enumeration of types. Every type that is a subtype of all enumerated types is a subtype of the compound type. All fields and methods of the enumerated types represent the fields and methods of the compound type. A compound type is a subtype of all of its enumerated types.

In [3] Compound Types are introduced to the programming language Java to provide a common type in a component-based system for all components that implement two independent specified interfaces. The approach of compound types is a “light weight construction to explore the middle ground between exclusive use of structure or name equivalence” ([3], p. 366). The here introduced extension of AspectJ represents some kind of light-weight version of compound types, because we restrict the applicability of compound types only to methods: Fields are not considered for the subtype relationship. But in contrast to the proposal in [3] we also permit to use structural types to occur within a compound type.

4. RELATED WORK

While structural type conformance is a known concept [19] which is applied in many functional programming languages like ML or Haskell, its usefulness in strongly typed object-oriented languages like Java is discussed in [15]. Löffers et al. extend the Java programming language in the same way as introduced above. However, as argued in section 3.2 pure structural types do not consider the semantics of certain interfaces: It is not possible to specify a structural type that states that the corresponding matching classes has to implement a certain interface. i.e. pure structural types do not consider the semantics of types that come from the nominal type system.

In [3] Compound Types are introduced to the programming language Java to provide a common type in a component-based system for all components that implement two independent specified interfaces. The approach of compound types is a “light weight construction to explore the middle ground between exclusive use of structure or name equivalence” ([3], p. 366). The here introduced extension of AspectJ represents some kind of light-weight version of compound types, because we restrict the applicability of compound types only to methods: Fields are not considered for the subtype relationship. But in contrast to the proposal in [3] we also permit to use structural types to occur within a compound type.

Parametric introductions [7] provide a very similar behavior without the need to extend the compiler: The algorithm of checking classes in respect to their structure and performing a corresponding type introduction can be specified on the application level. So, the computation of classes that match a certain structure can be shifted to the pointcut definition. The introduction can be defined in a way that it refers to the corresponding pointcut. Hence, it is possible to specify the algorithm performed by our compiler extension in order to introduce structural types to a corresponding pointcut definition.

In [6] Gybels and Brichau argue for the need of pattern-based crosscutting: pointcut languages should be able to reflect complex code patterns in order to specify at what join points an aspect should be woven. They also argue, that the current pointcut language of AspectJ does not permit to specify complex patterns. The here proposed application of structural and compound types are an additional way of creating more stable pointcuts.

Pointcut designators hasmethod and hasfield in AspectWerkz [2] respectively has and hasfield in JBoss [10] are added to support structural-based aspect specification. This permits to consider structural information within the pointcut specification. However, this selection does not address the interaction of objects whose structures are known. Our approach is to enhance the type system instead of using designators.

A related analysis of aspect-oriented mechanisms (procedures, annotations, pointcuts and advices) that may be used in modularizing crosscutting concerns is presented in [14]. This work suggests some guidelines to decide which mechanism should be used in which situation. However, there is no discussion on the suitability of those mechanisms in problematic situations presented in this paper – crosscutting that cannot be modularized due to typing conflicts.

Figure 8. RegisterMouseListener with compound types.

Figure 8 illustrates the use of a compound type for the aspect specification from the third example. Within aspect RegisterMouseListener the pointcut refers to objects of type [Applet, MouseListener, MouseMotionListener] and exposes an object of that type to the corresponding advice. All types that are subtypes of Applet, MouseListener and MouseMotionListener are subtypes of the compound type. Furthermore, the compound type is a subtype of each enumerated type. In Figure 8 there is one class MyApplet that extends Applet and implements the corresponding interfaces. Hence, MyApplet subclasses the compound type. All methods declared in Applet, MouseListener and MouseMotionListener represent methods of the compound type. Since the methods addMouseListener and addMouseMotionListener are defined in (a superclass of) Applet and the compound type is a subtype of MouseListener and MouseMotionListener, the advice can invoke the corresponding methods.

Compound types permit to specify pointcuts in a more expressive way than pure structural types because inheritance relationships between nominal types can be use as structural type information in

---

3 The term compound type is equivalent to intersection types [19]. However, since Java is the target language of [3] (which is the base language for AspectJ) we still use the term compound type.
5. DISCUSSION AND CONCLUSION

In this paper we argued that aspect-oriented systems need the ability to refer to types because of structural type information in addition to the nominal type information. We argued for this need by giving two examples which represent typical applications of aspect-oriented software development but which cannot be handled because of the underlying nominal type system. The given examples directly come from existing Java frameworks and are not artificially constructed. Based on this we discussed the applicability of structural and compound types. We illustrated that an aspect-oriented system based on a type system which provides structural and compound types in addition to nominal types permits to modularize the given crosscutting concerns.

We integrate the structural typing extension in AspectJST [1] an AspectJ extension that preserves the syntax of AspectJ. We use method signatures for matching where two methods are structurally matching if they have equal return and parameters types. Consequently, two types are structurally matching iff for every signature in the first type there is a structurally matching signature in the other type. For technical reasons we do not consider fields within the subtype relationship. For compound types, there is a restriction that among the types enumeration only one class is allowed (due to the Java class inheritance semantics).

The ideas of structural types and compound types are not new. The contribution of this paper is to show that there are a number of (even trivial) real-world examples where crosscutting occurs but which cannot be modularized by an aspect-oriented system such as AspectJ due to the underlying nominal type system. However, in untyped aspect-oriented systems like for example AspectS it is possible to modularize such concerns.

We think that from the aspect-oriented perspective the need for structural type information becomes more urgent than it might have been in the pure object-oriented world due to that aspects need to interact with objects of different types in a uniform way in the absence of a common (nominal) type available for them.

We emphasize that our intention is not to claim that structural correspondence should replace nominal types in general. We think that nominal types are an important way of expressing the conceptual model of the software system. Furthermore, we do not claim that structural typing should replace interface introductions in general. There are a number of successful applications of this idioms like for example [20] who add a persistency aspect to a number of classes.

There is relatively few works that analyze the relationship between type systems and aspect-oriented approaches, especially regarding the applicability of certain type systems for the purpose of aspect-oriented software development. We expect in the future a much wider discussion on the relationship between type systems and aspect-oriented systems because it seems as if both are closely related. On one hand current pointcut languages often rely on type-information. On the other hand, it looks like aspect-oriented system need to soften current nominal type systems because of the crosscutting characteristic of aspects, which permits to crosscut different types: the application of structural and compound types provides a step into this direction.

6. REFERENCES