Eco: A Flexible, Open and Type-Safe Framework for Aspect-Oriented Programming

Bruno De Fraine, Wim Vanderperren, Davy Suvée

Technical Report SSEL 01/2006/a
Vrije Universiteit Brussel

January, 2006

Abstract

In this report, we present the Eco aspect-oriented framework for Java. Eco's main goals are a flexible and open design and allowing static type-checking. Eco realizes its flexibility and openness by employing a pure and well-designed object-oriented approach. Static type-checking is realized by exploiting the novel Java generics feature. In this report, we evaluate Eco with respect to these goals and compare it to existing work.

Contents

1 Introduction **2**

2 The Eco Approach **4**
- 2.1 Invocations 4
- 2.2 Interceptors 5
- 2.3 Pointcuts 6
- 2.4 Deployment 8
- 2.5 Interaction Resolution 9

3 Evaluation **12**
- 3.1 Flexible Aspect Model 12
  - 3.1.1 Deployment of Aspectual Behavior through Instantiation 12
  - 3.1.2 Independent Pointcuts and Interceptors 13
- 3.2 Open Framework Approach 13
  - 3.2.1 First-class Deployment Entities 14
  - 3.2.2 No Global Static Aspects 14
  - 3.2.3 Base/Aspect Integration 15
- 3.3 Static Type Safety through Generic Typing 15

4 Related Work **17**

5 Conclusions and Future Work **19**

Chapter 1

Introduction

Aspect-Oriented Software Development (AOSD) [17] is a recent software engineering paradigm that aims at providing a better separation of concerns than possible using current paradigms. A good separation of concerns is crucial for realizing better comprehensible, reusable and maintainable software [22]. AOSD claims that by using classic approaches, certain concerns remain that cannot be confined into a single logical module. These concerns are called *crosscutting* because the concern virtually crosscuts the decomposition of the system. AOSD provides a solution to modularize these crosscutting concerns into a new entity named an *aspect*.

A wealth of approaches are currently available that provide support for aspect-oriented programming. They can be roughly divided in two categories: *framework-based approaches* and *language-based approaches*. Framework-based approaches, such as JBoss [5], PROSE [23] and AspectWerkz [3], introduce an aspect-oriented framework implemented in the base language to support aspect-oriented programming. Language-based approaches on the other hand, such as AspectJ [16], JAsCo [26] and CaesarJ [21], introduce a dedicated language or language extension for developing aspects. The obvious advantages of the typical framework approach are the compatibility with the existing development process and the easier acceptance of the approach. Dedicated aspect languages have the advantage of being more concise in their descriptions and are typically more expressive.

In this report, we propose a new framework-based AOSD approach for Java, named Eco, that aims at solving some of the deficiencies of current frameworks and that tries to further improve on their inherent advantages. The main design goal of Eco is to provide a *statically type-safe* framework for dynamic aspect-oriented programming that is both very *flexible* and *open*:

- *Static Aspectual Typing:* One of the main problems in current framework-based approaches is the lack of support for static aspectual typing, although the base language typically employs a statically typed language. Eco solves this by cleverly exploiting Java's generics feature [11].

- *Flexibility:* Eco aims to provide general solutions that do not limit any of the possibilities prematurely. This is not always the case in current framework-based approaches and even languages. For example, aspect deployment is typically very inflexible in that aspects are viewed as singleton entities that cannot be instantiated manually.

- *Openness:* A natural advantage of frameworks is that they are more open to adaptation and extension than language-based approaches because they are available as libraries on the same level as the base application. However, current aspect-oriented frameworks do not seem to pursue that goal. Aspect composition is for instance typically handled in a way that cannot be extended nor customized beyond some predefined solutions.

The presentation of Eco in this report is organized as follows: the next section provides an overview of aspect-oriented programming with the Eco framework. Afterwards, we present an in-depth evaluation of the framework that focuses on the three claimed properties. We then compare Eco to other state-of-the-art approaches that share one or more design goals. Finally, we present our conclusions and future work.

Chapter 2

The Eco Approach

This section presents an overview of the Eco approach from the point of view of the end developer. To modularize *crosscutting* concerns in an application, he/she can implement these concerns as aspectual behavior. This is done using a standard AOP mechanism where *pointcuts* select certain regions in the execution of the application (*join points*), and *interceptors* (or *advices*) specify behavior that is to be executed in addition to (or in place of) the standard behavior.

A graphical overview of the framework is given in figure 2.1. The four main parts of this model are discussed successively in the following sections.

2.1 Invocations

The `Invocation` type is an argument that the system passes to interceptors. It represents the intercepted join point. Join points are regions in the execution of the application, such as the execution of a method, access to a field, etc. An interceptor might require access to certain context information about the join point, e.g. the argument with which the method was invoked in case the join point is a method execution. The `getContext()` method *exposes* this kind of

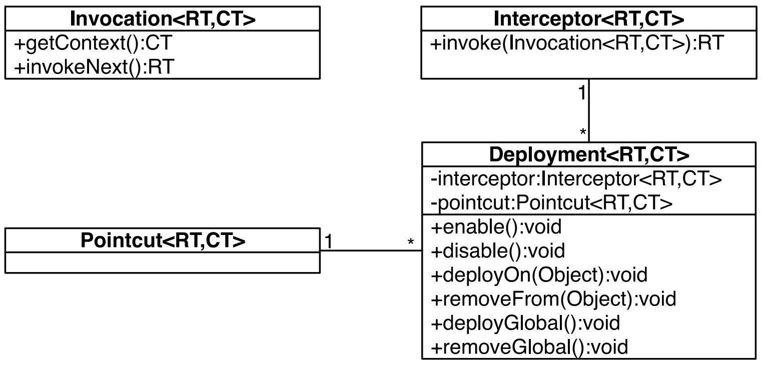

Figure 2.1: Overview of Eco approach.

```java
public class BeforeAfterInterceptor<RT,CT> implements Interceptor<RT,CT> {
    public final RT invoke(Invocation<RT,CT> invocation) {
        CT context = invocation.getContext();
        before(context);
        try { return invocation.invokeNext(); }
        finally { after(context); }
    }
    public void before(CT context) {}
    public void after(CT context) {}
}
```

Listing 1: A convenience interceptor for simple before/after advice.

context information: the interceptor calls it to retrieve a context object[1]. In addition to context information, the interceptor can also invoke the original behavior of the intercepted join point and retrieve its result. The `invokeNext()` method is available for this purpose, although its specification has to be generalized slightly to accommodate for the possibility of multiple interceptors that apply to the same join point: in that case, a chain of interceptors is formed according to a certain strategy (covered in section 2.5), and the `invokeNext()` method invokes the next interceptor in the chain (hence its name) or the original join point behavior when the interceptors have been exhausted.

Note that the `Invocation` type is parametrized by the generic type parameters `RT` (which represents the return type of the join point) and `CT` (which represents the type of the context object). These type parameters are carried over to the other core elements of the framework and always refer to the return type and context type of the intercepted join point.

2.2 Interceptors

An interceptor specifies the behavior that is to be applied at the intercepted join point. The `Interceptor` type captures this behavior in very general terms: it contains only an `invoke` method with the behavior to be used in place of the join point. This method takes an argument of type `Invocation` as input, and can use it to retrieve context information or to invoke the original behavior, if this is appropriate. There is one constraint however: as the normal behavior of the application has to continue after application of the interceptor, this method has to return an object of the same return type as the intercepted join point[2].

The developer implements aspectual behavior by subtyping the `Interceptor` type and specifying the behavior in the `invoke` method. Although this general `Interceptor` type is powerful (it can express any advice behavior of typical AOP approaches), it is not very convenient to specify basic aspectual behavior that is to be executed in addition to (i.e. *before* or *after*) the standard behavior of the join point. The Eco framework therefore provides a convenience class

[1]For simplicity, only the case of one context object is presented. See section 3.3 for a discussion on how to support a different number of context objects.

[2]Note that a generic return type does not conflict with methods that don't return a value. The type parameter `RT` can be bound to the "empty" type `java.lang.Void` to denote this case. Treating functions that do not return a value as a special case of functions that do return a value by bringing a unit type into existence is commonplace in functional languages that natively support parametric polymorphism, e.g. Haskell or Objective Caml.

```java
public class OrderTrace<RT> extends BeforeAfterInterceptor<RT,Order> {
    protected OutputChannel out;
    public OrderTrace(OutputChannel out) { this.out = out; }
    public void before(Order o) {
        out.println("Operation on " + o.getName());
    }
}
```

Listing 2: An interceptor for tracing order manipulations.

`BeforeAfterInterceptor` that is presented in listing 1. In this class, `invoke` is a *template method* [10] that retrieves the context information and executes the original behavior, but defers to the `before` and `after` methods for executing possible additional behavior. Besides convenience, this class has another advantage: when realizing aspectual behavior, a general *around* advice is more costly than e.g. an *after* advice [14]; as subtypes of `BeforeAfterInterceptor` only constitute simple before/after advices, the system can exploit this knowledge to realize them more efficiently behind the scenes. The general `invoke` method would then no longer be used, and invocations to the the `before` and `after` methods can be added directly to the original behavior.

Listing 2 illustrates how a developer can implement a simple interceptor to trace manipulations to `Order` objects from the problem domain of his/her application. As the type parameter `CT` is bound to the type `Order`, the advice behavior can invoke methods specific to this type (such as the `getName()` method) without having to resort to explicit casts. This implementation of the tracing interceptor encompasses a degree of customizability: the `OutputChannel` to which log messages are written can be specified as a parameter.

2.3 Pointcuts

Pointcuts are used to select the join points in the execution of the application that must be intercepted. A large number of pointcut approaches exist in the field of AOSD [12, 7, 8, 20], and Eco framework aims to be agnostic of a concrete pointcut model or language: different approaches could be integrated, possibly at the same time. Once defined, a pointcut is abstracted to the context type and the return type of the join point(s) it selects, at least in the viewpoint of the aspect developer (see figure 2.1); the system will of course need more knowledge to allow for its underlying machinery to intercept the selected join points. In this presentation we will use the pointcut language of the well-known AspectJ approach. When using the AspectJ pointcut model and its pointcut designators, there are still at least two possible methods of integration with the framework: either by using a dedicated language or by using an object-oriented representation of the language entities.

AspectJ itself uses a dedicated aspect language. This allows pointcuts to be written in a straightforward manner. To integrate this dedicated language in the Eco framework, we propose to provide the pointcut definition as an annotation to the class that represents this pointcut. These annotations are processed in a compile-time preprocessing step that parses the pointcut definition, sets up the required run-time information, and extracts and verifies the type information.

```java
@APointcut(
    "orderManip(Order o): execution(void Order.set*(*)) && this(o)"
)
public class OrderManip implements Pointcut<Void,Order> {}
```

Listing 3: A pointcut to select manipulations to `Order` objects.

```java
public class MethodExecutionExposingThis<RT,CT> implements Pointcut<RT,CT> {
    public MethodExecutionExposingThis(
        Class<CT> containingClass,
        Class<RT> returnType,
        String methodNameRegExp,
        Class... parameterTypes
    ) {
        ...
```

Listing 4: Signature of the Java representation of an execution pointcut that exposes the `this` object.

Listing 3 illustrates an example of a pointcut for selecting order manipulations using this method. The pointcut definition specifies to select the execution of methods with `void` return type and whose name starts with "set" on objects of class `Order`. Additionally, it exposes the involved `Order` object as a context object. Note that the `OrderManip` class is defined as a subtype of `Pointcut` with return type bound to `Void` and context type bound to `Order`; the preprocessor can verify that these types are compatible with the return type and context type of the join point(s) selected by the pointcut definition.

This integration of a dedicated language in an object-oriented framework leaves a number of things to be desired. Most importantly, it is not possible to build programmatic abstractions over the language entities: imagine a procedure that takes a class as an argument and that returns a pointcut to select the execution of the *setters* on objects of that class; this would allow to reduce the amount of boilerplate code in common pointcuts such as the above `OrderManip`. To enable these abstractions, we propose to use a Java representation of the entities of the pointcut language. This is illustrated in listing 4, which shows the signature of a class that represents the `execution` pointcut designator, together with the closely related functionality of exposing the object upon which the method was invoked[3]. With this object-oriented representation of one of the

[3]Note that the class literal expression `C.class`, where `C` is the name of a class, returns an object of type `Class<C>` that represents the class `C` [11]. The type parameter `C` allows for static type safety in reflective operations.

```java
static <CT> Pointcut<Void,CT> getClassSettersPC(Class<CT> theclass) {
    return new MethodExecutionExposingThis<Void,CT>(theclass, void.class, "set*");
}

Pointcut<Void,Order> orderManip = getClassSettersPC(Order.class);
```

Listing 5: A pointcut to select manipulations to `Order` objects, implemented using an auxiliary procedure to select the execution of the *setters* of a class.

```
1 Interceptor<Void,Order> ordertrace = new OrderTrace<Void>(System.err);
2
3 Pointcut<Void,Order> ordermanip = new OrderManip();
4
5 Deployment<Void,Order> deploy = new Deployment<Void,Order>(ordermanip,ordertrace);
6
7 deploy.deployGlobal();
```

Listing 6: Deployment of a tracing interceptor to a pointcut that selects manipulations to `Order` objects.

elements of the pointcut language, we can build the aforementioned procedure for setter pointcuts in listing 5, and use it in turn to create an `orderManip` pointcut (line 5) with exactly the same semantics as that of listing 3. The Java representation is a more tedious way to write pointcuts than the dedicated language, but the ability to write programmatic abstractions can compensate for this to a certain level: when using the `getClassSettersPC` procedure, the definition of `orderManip` pointcut (line 5) can be given on a higher level of abstraction because it hides the implementation details of the pointcut.

2.4 Deployment

In the Eco approach, interceptors and pointcuts are defined as independent and reusable entities. A separate deployment step is used to connect these two types of entities, and to activate the aspectual behavior. For this, the aspect developer uses the `Deployment` class. The constructor of this class expects the interceptor to be deployed, as well as the pointcut that selects at which join points the interceptor is to be applied. An example is shown in listing 6, where we install aspectual behavior to trace manipulations to `Order` objects on the error output stream. This is done by first instantiating the interceptor from listing 2 and the pointcut from listing 3. These instances are then used to create a `Deployment` object. Instances of `Interceptor` and `Pointcut` can be freely combined when creating a `Deployment`: for example, the same `OrderTrace` instance can also be used to create a deployment with another pointcut, or even with the same pointcut, in case it is desired to execute the aspectual behavior another time (possibly with different deployment options).

The deployment model of Eco supports different modes of deployment. The standard mode is to intercept join points regardless of the object in which it occurs. This is selected with the `deployGlobal()` method, as shown on line 7 of listing 6. Alternatively, the deployment can be limited to intercept only join points from specific join point objects. To this end, the `deployOn` method can be invoked with the object for which to activate interception as an argument (see figure 2.1). The aspectual behavior is active after one of these two deployment methods has been invoked. Eco is hence a very dynamic AOP approach [9], in the sense that it supports *hot deployment*: aspectual behavior can be added to (and removed from) an application at run-time, either with the deploy (and inverse remove) operations, or with the `disable()` and `enable()` methods that allow to switch off/on individual deployments without changing their deployment structure. In fact, the only way to activate aspectual behavior with Eco

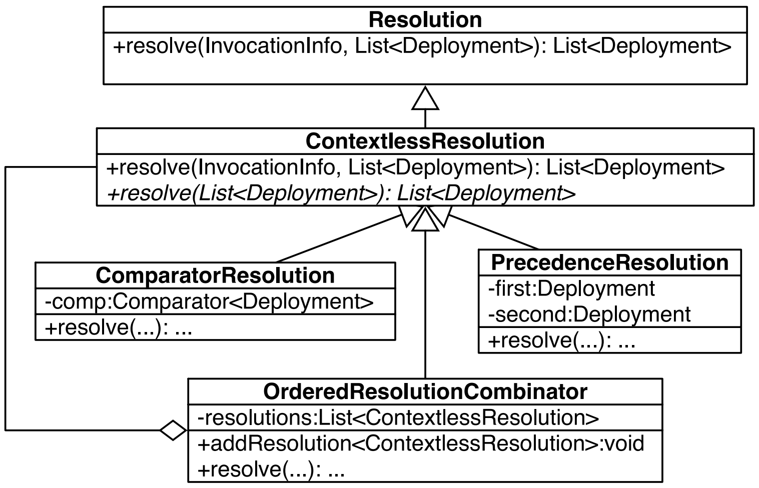
Resolution
+resolve(InvocationInfo, List<Deployment>): List<Deployment>
ContextlessResolution
+resolve(InvocationInfo, List<Deployment>): List<Deployment>
+resolve(List<Deployment>): List<Deployment>
ComparatorResolution
-comp:Comparator<Deployment>
+resolve(...): ...
PrecedenceResolution
-first:Deployment
-second:Deployment
+resolve(...): ...
OrderedResolutionCombinator
-resolutions:List<ContextlessResolution>
+addResolution<ContextlessResolution>:void
+resolve(...): ...

Figure 2.2: Organization of Interaction Resolution in Eco.

is through this run-time interface. However, if constant aspectual behavior is desired during the entire execution of the application, this can be simulated by placing the deployment code in a static initializer. The aspectual behavior is then deployed when the containing class is loaded.

2.5 Interaction Resolution

We define aspect interaction [24] as the situation where multiple deployments specify aspectual behavior for the same intercepted join point. The behavior in such a case is unspecified, so it is not clear which interceptions are applied, and in what order. The way in which the deployments are composed might not be of importance for the correct operation of an application (e.g., in case the interacting deployments have orthogonal functionality), but when the order of combination implies an important semantic difference for example, this decision does matter. In such a case, one needs to be able to influence the resulting composition, i.e. it must be possible to control how the interaction is resolved.

The solution that Eco provides for the resolution of interactions is shown in figure 2.2. It follows the same design strategy as the definition of interceptors in section 2.2. At the root is a very flexible `Resolution` type that contains a single `resolve` method. At every possible join point, this method is called with a representation of the join point and the list of *triggered* deployments (i.e. active deployments whose pointcut matches the intercepted join point). The method then returns a list of those deployments whose aspectual behavior must be applied, in the order in which they must be applied[4]. As such, resolution strategies have full control over the aspectual behavior: they can effectively reorder the deployments, remove triggered deployments and/or add untriggered deployments, and this resolution can vary over different join points. It is clear however that

[4]Note that the `resolve` method is queried for this information even if no deployments originally triggered.

```java
Deployment<Void,Order> tracing = ...
Deployment<Void,Order> synchro = ...
Deployment<Void,Order> persist = ...

OrderedResolutionCombinator globalres = new OrderedResolutionCombinator();
globalres.addResolution(new PrecedenceResolution(synchro, persist));
globalres.addResolution(new PrecedenceResolution(tracing, persist));

System.installGlobalResolution(globalres);
```

Listing 7: Installation of a combined resolution strategies regarding the persistence concern.

this flexibility has a high performance cost, as the resolution strategy has to be queried at each possible join point and its result cannot be cached. The system therefore provides a subclass `ContextlessResolution` which defines the general `resolve` method in terms of an abstract `resolve` method that does not depend on the join point argument and that adheres to a contract of being purely functional (i.e. having the same return value upon each invocation with the same argument values). This subclass is not as much a convenience class as an opportunity for the system to optimize the implementation of typical resolutions: when the installed resolution strategy is a `ContextlessResolution`, the system is expected to optimize the resolution internally by caching the results of the resolution strategy.

Eco further provides a number of predefined resolution strategies. The class `PrecedenceResolution` implements a simple resolution that ensures that the aspectual behavior of two given deployments is applied in a certain order when they are triggered at the same join point (it realizes this by swapping the position of the deployments if they occur in incorrect order). Alternatively, `ComparatorResolution` uses a standard Java comparator to order the triggered deployments. Other standard resolutions are of course imaginable, e.g. mutual exclusion between deployments or precedence relations between deployed interceptors instead of specific deployments. Furthermore, a developer can always implement a custom resolution strategy by subclassing the `ContextlessResolution` or `Resolution` type.

A more fundamental property of resolutions in Eco, is that it is possible to combine several resolutions using combinators. The `OrderedResolutionCombinator` is an example of such a combinator. Several resolutions can be registered with an instance of this class using the `addResolution` method. When the combinator is then asked to resolve a list of deployments, it will pipe this list through each registered resolution (in order of registration), passing the output of the first as input to the second, and so on. An example of its usage is shown in listing 7. We image that three concerns are deployed as aspectual behavior: tracing, synchronization and data persistence. In order to minimize the risk of data loss, we want the persistence interceptor to appear closest to the original join point in the interceptor chain (so its *after* advice will be executed immediately after the original behavior), i.e. after the other concerns in the list of deployments; other than that, we assume that the relative order of synchronization and tracing is not of direct importance. This behavior is accomplished in the example by configuring an ordered combinator with two precedence resolutions and installing this combined resolution as the global resolution strategy. The combined effect will be that, if triggered, the persistence deployment will be placed after either two of the other concerns (both when they occur together or separately) in the list of resolved deployments. When the persistence deployment was not triggered, the order will not be modified.

Chapter 3

Evaluation

In this section, we present an evaluation of the proposed approach. The purpose of this evaluation is to state the problems that motivated the creation of Eco, and to explain the rationale behind its design decisions. It also tries to clarify how Eco relates to mainstream approaches in the field of AOSD. The discussion is organized according to the three claimed properties of Eco: flexibility, openness and static type-safety.

3.1 Flexible Aspect Model

This section focuses on the structural design decisions that underpins the model of Eco. These decisions relate primarily to the flexible use and reuse of entities in the aspect model.

3.1.1 Deployment of Aspectual Behavior through Instantiation

The ability to deploy aspectual behavior in different contexts has been recognized as an important reuse property for aspect-oriented technologies [13, 21, 26]. The AspectJ approach organizes this reuse through the concept of abstract aspects. Such aspects define advice methods for yet unspecified (abstract) pointcuts, and subaspects are subsequently used to deploy the aspect by filling in the abstract pointcuts. An alternative model consists in using the concept of instantiation to deploy aspectual behavior, and to pass in the concrete pointcuts as instantiation parameters. AspectJ precludes this possibility by stipulating that its aspects are always implicitly instantiated by the system, but technologies such as JAsCo or JBoss do take this approach as their deployment of aspects through *connectors* or *XML deployment descriptors* constitutes an instantiation (or can conceptually be considered one).

Eco follows the latter design and allows aspectual behavior to be deployed through the creation of a `Deployment` instance, without requiring the definition of a new subtype. The first reason for this is that we consider it a limitation of object-oriented possibilities to exclude instantiation of aspects as a variation point besides inheritance. Put differently, we encourage having different instances of the same aspect, instead of having different subaspects with one

```java
new Deployment<Void,Order>(
   new MethodExecutionExposingThis<Void,Order>(Order.class, void.class, "finish"),
   new BeforeAfterInterceptor<Void,Order>() {
      public void after(Order o) { DataStore.saveOrder(o); }
   }
).deployGlobal();
```

Listing 1: Direct complete specification of aspectual behavior to persistently store orders after they are finished.

instance. The second reason is that the inheritance structure is statically fixed in Java; if the deployment of aspects requires the creation of a new subtype, then it is impossible to vary the aspectual behavior based on run-time values (e.g. imagine deploying as many instances of an aspect as an integer argument to the program prescribes). Instantiation does not suffer from this limitation. Note that this latter point is related to the discussion of dynamic deployment in the next section.

3.1.2 Independent Pointcuts and Interceptors

In addition to the reuse of interceptor behavior, Eco also aims at providing equally flexible reuse of pointcut definitions. In the JAsCo approach for example, it is not possible to share pointcuts between different aspect entities. This hampers the possible reuse of pointcut definitions. Eco on the other hand employs a straightforward design where pointcuts and interceptors are defined independently, and where an interceptor can be freely deployed to a pointcut as long as the respective type parameters are not in conflict. This allows to reuse both a pointcut and an interceptor definition, even if they have been independently developed. As a consequence, Eco differs from most AOP approaches in that it does not couple the definition of pointcuts and interceptors to an aspect entity, although pointcuts and interceptors classes can be grouped in a package if they are logically related.

Note however that Eco does not enforce this separate definition of pointcuts and interceptors. If reuse is considered unlikely and the overhead of separate class and variable definitions is deemed too high, a complete specification of aspectual behavior can be given in one statement by inlining the definitions. This is shown in listing 1, where we implement a complete aspect to write `Order` objects to a data store after their `finish()` method has been invoked. Apart from `Order` and `DataStore`, no types are used that are not directly present in the Eco framework. The interceptor is defined as an anonymous class.

3.2 Open Framework Approach

Eco can be called a Java AOP *framework* primarily because the specification and deployment of aspectual behavior occurs in plain Java code. As a consequence, integration with existing Java tools is facilitated and developers do not have to learn another language to employ Eco. But although these are handy advantages, they do not constitute the main reasons why Eco proposes to offer aspect-oriented facilities at the level of a run-time Java API. The main motivations are that the aspectual behavior is better integrated with the base application (i.e. the code which the aspects advice), and that it allows for powerful programmatic abstractions.

3.2.1 First-class Deployment Entities

Most of the current aspect-oriented languages and frameworks do not allow to deploy aspect using a plain Java API[1]. Instead, common options are dedicated languages (e.g. AspectJ and JAsCo), XML configuration files or code annotations (e.g. JBoss and AspectWerkz) for deployment. Although these solutions are generally more straightforward to employ, the inherent downside is that these deployment descriptions no longer have the same status as regular code objects with regard to abstraction, run-time construction, etc. We can say that the deployment entities are not first-class. As a consequence, for example, there are no suitable means to avoid the duplication between similar (or largely similar) deployments. Eco specifically targets first-class deployment entities and therefore deployment occurs primarily through a Java API. Although this is more tedious at first, this can be compensated by the ability to abstract detail in higher-lever deployment strategies. (Note the similarity with the discussion on the merits of a Java representation of pointcuts in section 2.3.) It is for example possible with Eco to declare a procedure that takes an `Interceptor` as an argument and that deploys this interceptor with a particular pointcut and an appropriate resolution strategy. This procedure can then be shared between the deployment of several interceptors.

3.2.2 No Global Static Aspects

Aspect-oriented programming historically departed from a view where aspects specify their behavior as rules over the compile-time structure of an application (*"whenever method `Foo()` is called, write a log entry"*). This corresponds with a model of global and static aspects, i.e. aspects that have but one instance that applies to all objects in the base application, at all times. More recently, a trend towards dynamic aspects [9] and association aspects [25] has proven this view to be too simplistic. To a certain extent, we notice that the process of making aspects dynamic and non-global involves giving them abilities of standard classes (multiple instances, run-time construction, . . .). By already organizing all aspectual behavior in terms of classes and objects, Eco can natively meet these requirements.

More concretely, the term *dynamic aspects* generally refers to aspectual behavior that is triggered on run-time conditions or that can be recomposed at run-time. The former concern is primarily related to the employed join point model and its expressiveness; it is therefore not directly addressed by Eco. It certainly meets the latter concern however. As explained in section 2.4, deployment and undeployment is organized entirely dynamic through the API of Eco. *Association aspects* on the other hand, refers to aspects with different instances for different objects or object groups. Eco accommodates for this by making the creation of `Interceptor` and `Deployment` instances explicit and controllable by

[1]JBoss is a notable exception that also offers a run-time API for deployment, however primarily with the objective of enabling run-time deployment.

the aspect developer. This allows him/her to flexibly set up aspect instances and associate them with objects of the base application.

3.2.3 Base/Aspect Integration

Asymmetric AOP approaches make a strong distinction between the *base* application that implements the main or business concerns, and *aspects* that implement crosscutting concerns. Although the Eco approach defines aspectual behavior in regular classes, it is still of asymmetric nature: interceptors are for example clearly distinguishable from base code as subtypes of `Interceptor`, and interceptor code is clearly aware of its status (e.g. when it invokes the original join point behavior).

Eco's usage of plain code deployment does however facilitate the integration between the aspects and the base application. Aspects may require parameters from the base application for their operation that are not directly available at the intercepted join point. Concrete examples could be the handle of the application's log file in case of a tracing aspect, or the observers to notify in case of an aspect that intercepts changes to an observed subject. A first solution for this problem could be that the aspect implements additional aspectual behavior to capture the required information. This is not straightforward and error-prone: e.g., one has to verify that in all cases this capture has succeeded before the main aspectual behavior of the aspect is triggered. A second solution involves making information available for static lookup: either the application makes the desired objects available through static fields or methods, or the aspects are retrieved by the base application through for instance AspectJ's static `aspectOf()` construct, after which the base application *injects* the information into the aspects. Solutions involving a static lookup impose a restriction as the information must be addressable with a unique key. Eco can make this integration more flexible, as the aspectual behavior can be deployed from the point where the parameter information is normally available in the base code. It can there be easily passed in as a parameter to the relevant interceptor instance (as is done for the `OutputStream` parameter in line 1 of listing 6, for example).

3.3 Static Type Safety through Generic Typing

Java is a programming language with static nominal type checking of explicitly declared types and exceptions. Irrespective of discussions on the most productive or safe type systems, it seems logical for an aspect-oriented extension of Java to use the same type system as the base language. Dedicated aspect languages such as AspectJ or JAsCo clearly aim to provide this, but Java aspect frameworks such as JBoss or Spring/AOP typically reduce this to run-time type checking. For example, context parameters are passed to advice blocks as arguments of the general type `Object` and have to be explicitly casted to the assumed type, which leads to run-time errors if the advice is deployed with an incompatible pointcut. It is one of the contributions of Eco to implement an AOP framework with hot deployment capabilities without sacrificing Java's static type safety.

To realize static type-safety in a framework approach, Eco relies on the generic typing feature made available in Java 5. Although we demonstrate that this allows to check type compatibilities between the major parts of the framework, we did encounter limitations to this approach. For example, it is not possible to work with a variable number of type parameters, which is problematic to support invocations with multiple context objects. Possible workaround solutions are to group the context objects in one container object, or to define different versions of the major classes of the framework for the different possible numbers of context objects. A more fundamental solution seems to require a smarter type system than what is currently available in Java.

Chapter 4

Related Work

Nowadays, several aspect-oriented technologies are emerging, which aim at providing flexible and open-ended aspect-oriented technologies.

AspectS [15] is a general-purpose, aspect-oriented framework for Smalltalk. Similar to Eco, AspectS exploits the expressive power of the existing base language encapsulation mechanisms for modularizing pointcuts, advices and their corresponding deployments. At the pointcut level, AspectS offers a more open design compared to Eco, as its pointcuts can be build directly on top of Smalltalk's extensive meta-programming facilities. Eco on the other hand, offers only a limited set of primitives that however should suffice in most cases to build up other, more involved pointcut designators. The main advantage of Eco over AspectS however, is that Eco allows for the independent specification of both pointcuts and advices, whereas AspectS inlines the definition of both within the aspect deployment, hence obstructing possible reuse. Finally, in contrast to Java, Smalltalk does not support the notion of static type safety. Inherently, this distinction is also reflected at the AOP framework level, as AspectS does not support static type-safety for aspects in contrast to Eco.

Josh [6] and the AspectBench Compiler (*abc*) [2] represent efforts to build open aspect systems at the compiler level. Josh provides an AspectJ-like language built on top of the compile-time reflection library Javassist. New pointcut designators and generic advice descriptions can be implemented as Java code that uses the Javassist API and that is executed by the Josh weaver. Similarly, *abc* is a reimplementation of the AspectJ compiler using the Polyglot and Soot frameworks (for frontend and backend respectively). It also aims to enable easy implementation of extensions such as new language features or implementation techniques, but features static checks in the language frontend in addition to Josh. Both approaches focus on building extensions to dedicated AOP languages by providing access to lower-level compiling and weaving techniques, whereas Eco primarily aims to allow the developer to build abstractions on developer-level primitives.

Eco focuses on providing an open, aspect-oriented framework that can easily be extended in order to conform to the requirements of a particular developer. However, the general-purpose nature of Eco could sometimes obstruct its ease of use, whereas domain-specific aspect languages are generally more expressive and understandable. Nowadays, the combined use of separately defined domain-specific aspect languages is troublesome, as each language typically employs its own set of conflicting integration mechanisms. Therefore, *Pluggable AOP* [18] aims at offering a semantical framework in which independently developed aspect languages can be composed and collaboratively work together. For this, Pluggable AOP introduces a mixin-based approach, where a so called *aspect mixin mechanism* transforms the base mechanism description and introduces some additional description that can be understood by other, independently specified aspect mixins that are described in the same formal model.

Similar to Pluggable AOP, Reflex [27] aims at providing a framework that allows for a straightforward collaboration between domain-specific aspect languages. To this end, Reflex introduces the concept of a versatile AOP kernel that acts as a base to which all domain-specific aspect languages are translated. The notions introduced by the Reflex AOP kernel bear quite some similarities with the ones proposed by Eco: both pointcuts (modeled as *hooksets* which are similar to Eco's *pointcuts*) and advices (modeled as metaobject classes) are encapsulated as independent entities that are later on explicitly combined employing so called *link* bindings. A Reflex link is similar to an Eco deployment. Additionally, Reflex supports interaction resolution by building up composition operators out of lower level kernel operations. In contrast with Eco however, Reflex allows to automatically detect interaction conflicts, which can be automatically resolved by attaching the necessary link interaction selectors. Eco on the other hand allows to describe aspects on a higher level in contrast to Reflex, from which the lower-level mechanisms are only intended to be observed and employed by the implementor of a domain-specific aspect language.

Chapter 5

Conclusions and Future Work

We present the Eco framework that aims to improve on current state-of-the-art aspect-oriented frameworks. The Eco framework especially excels at providing a statically type-safe approach to dynamic AOP while being very flexible and open. An in-depth evaluation reveals that Eco is able to provide these benefits thanks to a well-designed object-oriented model that allows for definition and deployment in plain code and that employs Java's generics feature throughout.

In order to enable the Eco approach we are currently working on an implementation on top of BEA's novel JRockit AOP-enabled virtual machine [4]. This virtual machine supports a low-level API for AOP and aims to provide aspectual behavior at very little performance cost. A first proof-of-concept prototype of Eco shows promising results.

Bibliography

[1] Mehmet Akşit, editor. *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003).* ACM Press, March 2003.

[2] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhotak, Ondrej Lhotak, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. In Peri Tarr, editor, *Proc. 4rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2005)*, pages 87–98. ACM Press, March 2005.

[3] Jonas Bonér and Alexandre Vasseur. AspectWerkz: simple, high-performant, dynamic, lightweight and powerful AOP for Java. Home page at `http://aspectwerkz.codehaus.org/`, 2004.

[4] Jonas Bonér, Alexandre Vasseur, and Joakim Dahlstedt. JRockit JVM support for AOP, part 1. Technical report, BEA dev2dev, August 2005.

[5] Bill Burke et al. JBoss Aspect-Oriented Programming. Home page at `http://www.jboss.org/products/aop`, 2004.

[6] Shigeru Chiba and Kiyoshi Nakagawa. Josh: an open AspectJ-like language. In Lieberherr [19], pages 102–111.

[7] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In A. Yonezawa and S. Matsuoka, editors, *Metalevel Architectures and Separation of Crosscutting Concerns 3rd Int'l Conf. (Reflection 2001), LNCS 2192*, pages 170–186. Springer-Verlag, September 2001.

[8] Rémi Douence, Olivier Motelet, and Mario Südholt. Sophisticated crosscuts for e-commerce. In Lodewijk Bergmans, Maurice Glandrup, Johan Brichau, and Siobhán Clarke, editors, *Workshop on Advanced Separation of Concerns (ECOOP 2001)*, June 2001.

[9] Robert E. Filman, Michael Haupt, and Robert Hirschfeld, editors. *Dynamic Aspects Workshop*, March 2005.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[11] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison-Wesley Professional, third edition, June 2005.

[12] Kris Gybels and Johan Brichau. Arranging language features for pattern-based crosscuts. In Akşit [1], pages 60–69.

[13] Stefan Hanenberg and Rainer Unland. Using and reusing aspects in AspectJ. In Kris De Volder, Maurice Glandrup, Siobhán Clarke, and Robert Filman, editors, *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001)*, October 2001.

[14] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In Lieberherr [19], pages 26–35.

[15] Robert Hirschfeld. Aspect-oriented programming with AspectS. In Mehmet Akşit and Mira Mezini, editors, *Net.Object Days 2002*, October 2002.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[17] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[18] Sergei Kojarski and David H. Lorenz. Pluggable AOP: Designing aspect mechanisms for third-party composition. In *Proceedings of the OOPSLA '05 conference*, pages 247–263. ACM Press, 2005.

[19] Karl Lieberherr, editor. *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*. ACM Press, March 2004.

[20] Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote pointcut: a language construct for distributed AOP. In Lieberherr [19], pages 7–15.

[21] Klaus Ostermann and Mira Mezini. Conquering aspects with Caesar. In Akşit [1], pages 90–99.

[22] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–1058, December 1972.

[23] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In Gregor Kiczales, editor, *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, pages 141–147. ACM Press, April 2002.

[24] Elke Pulvermüller, Andreas Speck, James Coplien, Maja D'Hondt, and Wolfgang De Meuter. Feature interaction in composed systems. In *ECOOP '01: Proceedings of the Workshops on Object-Oriented Technology*, pages 86–97, London, UK, 2002. Springer-Verlag.

[25] Kouhei Sakurai, Hidehiko Masuhara, Naoyasu Ubayashi, Saeko Matsuura, and Seiichi Komiya. Association aspects. In Lieberherr [19], pages 16–25.

[26] Davy Suvée and Wim Vanderperren. JAsCo: An aspect-oriented approach tailored for component based software development. In Akşit [1], pages 21–29.

[27] Éric Tanter and Jacques Noyé. A versatile kernel for multi-language AOP. In Robert Glück and Michael R. Lowry, editors, *Generative Programming and Component Engineering, 4th International Conference (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 173–188. Springer, September 2005.