Tel Aviv University
Raymond and Beverly Sackler Faculty of Exact Sciences
School of Computer Science

# Automatic Tool for Refactoring Aspects Away

This thesis is submitted as fulfillment of the requirements towards the M.A degree in the School of Computer Science, Tel Aviv University

by

Michael Kleyman

The work on this thesis was carried out under the supervision of Prof. Amiram Yehudai and Dr. Shmuel Tyszberowicz

October 2005

# 1. Introduction

The agile development approach welcomes changing requirements even late in the development process [1]. Refactoring is a major technique used to cope with changes. It is a process and a set of techniques to reorganize code while preserving the external behavior of a working system [2]. Several commercial tools that can perform this process automatically are available. Among these tools are the Eclipse IDE for Java [3] and ReSharper Visual Studio add-in for C# [4].

Aspect Oriented Programming (AOP) is another powerful technology that allows easily introducing changes to a software system. Sometimes, though aspects have been used in the programming process, they cannot be employed in production software, for several reasons. For example, the policy of the organization may prohibit using aspects in production, because AOP is not considered mature yet, or some requirements are fulfilled better with pure object-oriented rather than aspect-oriented code. A typical case is adding resource pooling to an existing program.

This work describes ACME, a tool that implements a refactoring process for cases when the AspectJ code should not remain in the system. It allows a programmer to automatically convert AspectJ code to pure Java source code. The goal of this tool is to create modular and readable Java code, as opposed to the AspectJ compiler, which produces bytecode.

ACME offers a solution for an issue that has not been dealt with before. Most work on aspect-oriented refactoring focuses on introduction focuses on introduction of aspects into existing systems. S. Rura in [13] discusses refactoring manipulations that will be valid in an aspect-oriented environment. His work includes discussion of regular object-oriented refactorings in such an environment, as well as new manipulations, specific to aspect-oriented languages. Rura's work deals with atomic operations, such as moving an introduced member from an aspect to the class into which the member is introduced. In this work, I show common situation in which a programmer may want to remove aspects from a system and present a system that automatically performs this removal.

The thesis is organized as follows. Section 2 describes the AspectJ language in detail. Motivation for developing ACME is explained in section 3. Examples of supported transformations are shown in section 4. Section 5 explains design decisions made. In section 6 I discuss which features of the AspectJ languages can be supported by

ACME. A real-life situation in which ACME could have been helpful is described in section 7. Section 8 shows possible extensions of ACME and presents the conclusions from this work.

# 2. Aspect-Oriented Programming

The main purpose of Aspect-Oriented Programming (AOP) is to enable proper encapsulation of crosscutting concerns. One of the advantages of object-oriented languages is the modularity they provide. Strong cohesion is an important design issue, which requires that each module will be handling only one concern. It is also important that each concern will only be handled in one module only. Nevertheless, there are cases where a concern is implemented in many modules; i.e. it crosscuts the modules and are not implemented in a single class, but rather scattered in several classes. This scattering may be unavoidable either due to the secondary role of these concerns, or because their nature requires dealing with them in many different locations. Another problem is the case where one module is dealing with more than one concern. This phenomena is called tangling code.

Aspects are believed to be the solution to both of these problems. The crosscutting or tangling code is represented in aspects. Each aspect deals with one concern. Aspects look like classes with additional information of where the crosscutting code, which has been removed from the classes in order to provide good modularity, has to be weaved into the source code.

This weaving is a new mechanism that is taken care of by special compilers for AOP languages. Weaving allows the programmer to write a portion of code and declare when this code should be executed in the course of execution of other modules.

The rest of this section describes the AspectJ language in detail. It is based on the book by R. Laddad [5].

## *2.1. AspectJ*

In AspectJ, the implementation of the weaving rules by the compiler is called crosscutting. This name is due to the fact that the weaving rules *cut across* multiple modules in a systematic way in order to modularize the crosscutting concerns. AspectJ defines two types of crosscutting: *static* crosscutting and *dynamic* crosscutting.

### 2.1.1. Dynamic crosscutting

Dynamic crosscutting is the weaving of new behavior into the execution of a program. Most of the crosscutting that happens in AspectJ is dynamic. Dynamic crosscutting

augments or even replaces the core program execution flow in a way that cuts across modules, thus modifying the system behavior. For example, if you want to specify that a certain action be executed before the execution of certain methods or exception handlers in a set of classes, you can just specify the weaving points and the action to take upon reaching those points in a separate module. This specification is made using *pointcuts* and *advices*, as will be explained later.

### 2.1.2. Static crosscutting

Static crosscutting is the weaving of modifications into the static structure—the classes, interfaces, and aspects—of the system. By itself, it does not modify the execution behavior of the system. The most common function of static crosscutting is to support the implementation of dynamic crosscutting. For instance, you may want to add new data and methods to classes and interfaces in order to define class-specific states and behaviors that can be used in dynamic crosscutting actions. Another use of static crosscutting is to declare compile-time warnings and errors across multiple modules.

### 2.1.3. Crosscutting elements

AspectJ uses extensions to the Java programming language to specify the weaving rules for the dynamic and static crosscutting. The AspectJ extensions use the following constructs to specify the weaving rules programmatically; they are the building blocks that form the modules that express the crosscutting concern's implementation.

#### 2.1.3.1.Join point

A *join point* is an identifiable point in the execution of a program. In AspectJ, everything revolves around join points, since they are the places where the crosscutting actions are woven in. The most commonly used join points are method calls and assignments to a member of an object. A join point may also be a constructor invocation, exception handler execution, class or object initialization and advice execution.

There are two types of join points for method and constructor invocations: execution and call join points. The execution join point is in the method body itself, whereas the call join points are in other parts of the program, which are usually the methods that

are calling this method. For most purposes, the difference between the execution and call join points does not matter; however, there are subtle differences, some of which will be discussed in detail in Section 5.

All join points also have a context associated with them. For example, a call to a join point in a method has the caller object, the target object, and the arguments of the method available as the context. Similarly, the exception handler join point would have the current object and the thrown exception as the context. This context can be used in the crosscutting code, as will be explained below.

#### 2.1.3.2.Pointcut

A *pointcut* is a program construct that selects join points and collects context at those points. There are two ways that pointcuts match join points in AspectJ. The first way captures join points based on the category to which they belong, such as method call join points or field get join points. The pointcuts that map directly to these categories or *kinds* of exposed join points are referred as *kinded* pointcuts.

The second way that pointcut designators match join points is when they are used to capture join points based on matching the circumstances under which they occur, such as control flow, lexical scope, and conditional checks. These pointcuts capture join points in any category as long as they match the prescribed condition. Some of the pointcuts of this type also allow the collection of context at the captured join points. Let's take a more in-depth look at each of these types of pointcuts.

**Kinded pointcuts** follow a specific syntax to capture each kind of exposed join point in AspectJ. For example, to capture all write accesses to a private _balance field of type float in the Account class, you would use a set() pointcut as follows:

```
set(private float Account._balance)
```

At times, a pointcut will specify a join point using one particular signature, but often it identifies join points specified by multiple signatures that are grouped together using matching patterns. For example, the pointcut

```
public void Account.set*(*)
```

matches all public methods in the Account class with a name starting with set and taking a single argument of any type;

```
* Account.*(..)
```

matches all methods in the Account class including even methods with private access;

```
public void Account+.set*(int)
```

matches all public methods in the Account class and its subclasses with a name starting with set and taking a single integer argument and

```
* javax..*.add*Listener(EventListener+)
```

matches any method whose name starts with `add` and ends in `Listener` in the `javax` package or any of its direct and indirect subpackages that take one argument of type `EventListener` or its subtype, such as

```
TableModel.addTableModelListener(TableModelListener)
```

Such patterns may be used in other types of pointcuts as well. They are commonly referred to as *type patterns.*

**Control-flow based pointcuts**

These pointcuts capture join points based on the control flow of join points captured by another pointcut. The control flow of a join point defines the flow of the program instructions that occur as a result of the invocation of the join point. A control-flow pointcut always specifies another pointcut as its argument. There are two control-flow pointcuts. The first pointcut is expressed as `cflow(Pointcut)`, and it captures all the join points in the control flow of the specified pointcut, including the join points matching the pointcut itself. The second pointcut is expressed as `cflowbelow(Pointcut)`, and it excludes the join points in the specified pointcut.

**Lexical-structure based pointcuts**

A lexical scope is a segment of source code. It refers to the scope of the written code, as opposed to the scope of the code when it is being executed, which is the dynamic scope. Lexical-structure based pointcuts capture join points occurring inside a lexical scope of specified classes, aspects, and methods. There are two pointcuts in this category: `within()` and `withincode()`. The `within()` pointcuts take the form of `within(TypePattern)` and are used to capture all the join points within the body of the specified classes and aspects, as well as any nested classes. The `withincode()` pointcuts take the form of either `withincode(MethodSignature)` or `withincode(ConstructorSignature)` and are used to capture all the join points inside a lexical structure of a constructor or a method, including any local classes in them.

One common usage of the `within()` pointcut is to exclude the join points in the aspect itself. For example, the following pointcut excludes the join points

corresponding to the calls to all print methods in the `java.io.PrintStream` class that occur inside the `TraceAspect` itself:

```
call(* java.io.PrintStream.print*(..)) && !within(TraceAspect)
```

**Execution object pointcuts**

These pointcuts match the join points based on the types of the objects at execution time. The pointcuts capture join points that match either the type of the current object, or the target object, which is the object on which the method is being called. Accordingly, there are two execution object pointcut designators: `this()` and `target()`.

In addition to matching the join points the same syntax is also used to collect the context at the specified join point. This allows the advice body to refer to the object on which the join point was matched. In this case, usually, the execution object pointcut will match all join points.

The `this()` pointcut takes the form `this(`*`Type`* or *`ObjectIdentifier`*`)`; it matches all join points that have a this object associated with them that is of the specified type or the specified *`ObjectIdentifier`*'s type. In other words, if you specify *Type*, it will match the join points where the expression `this instanceof <`*`Type`*`>` is true. The form of this pointcut that specifies *`ObjectIdentifier`* is used to collect the `this` object. If you need to match without collecting context, you will use the form that uses *`Type`*, but if you need to collect the context, you will use the form that uses *`ObjectIdentifier`*.

The `target()` pointcut is similar to the `this()` pointcut, but uses the target of the join point instead of this. The `target()` pointcut is normally used with a method call join point, and the target object is the one on which the method is invoked. A `target()` pointcut takes the form `target(`*`Type`* or *`ObjectIdentifier`*`)`.

**Argument pointcuts**

These pointcuts capture join points based on the argument type of a join point.

For method and constructor join points, the arguments are simply the method and constructor arguments. For exception handler join points, the handled exception object is considered an argument, whereas for field write access join points, the new value to be set is considered the argument for the join point. Argument-based pointcuts take the form of `args(`*`TypePattern`* or *`ObjectIdentifier`*`, ..)`.

Similar to execution object pointcuts, these pointcuts can be used to capture the context.

**Conditional check pointcuts**

This pointcut captures join points based on some conditional check at the join point. It takes the form of `if(BooleanExpression)`.

#### 2.1.3.3.Advice

Advice is the action and decision part of the crosscutting puzzle. It is a method-like construct that provides a way to express crosscutting action at the join points that are captured by a pointcut. The three kinds of advice are as follows:

- *`Before`* advice executes prior to the join point.
- *`After`* advice executes following the join point.
- *`Around`* advice replaces the join point's execution. This advice is special in that it has the ability to bypass execution, continue the original execution, or cause execution with an altered context.

An advice can be broken into three syntactical parts: the advice declaration, the pointcut specification, and the advice body. This can be seen on the next example.

```
before(Connection connection): connectionOperation(connection) {
    System.out.println("Performing operation on " + connection);
}
```

The part before the colon is the advice declaration, which specifies when the advice executes relative to the captured join point—before, after, or around it. The advice declaration also specifies the context information available to the advice body, such as the execution object and arguments, which the advice body can use to perform its logic in the same way a method would use its parameters. It also specifies any checked exceptions thrown by the advice.

The part after the colon is the pointcut; the advice executes whenever a join point matching the pointcut is encountered.

The advice body is similar to a method body. It contains the actions to execute and is within the {}.

**The before advice**

The before advice executes before the execution of the captured join point. In the following code snippet, the advice performs authentication prior to the execution of any method in the Account class:

```
before() : call(* Account.*(..)) {
    ... authenticate the user
}
```

If you throw an exception in the before advice, the captured operation won't execute. For example, if the authentication logic in the previous advice throws an exception, the method in Account that is being advised won't execute.

**The after advice**

The after advice executes after the execution of a join point. Since it is often important to distinguish between normal returns from a join point and those that throw an exception, AspectJ offers three variations of after advice: after returning normally, after returning by throwing an exception, and returning either way.

**The around advice**

The `around` advice surrounds the join point. It has the ability to bypass the execution of the captured join point completely, or to execute the join point with the same or different arguments. It may also execute the captured join points multiple times, each with different arguments.

If within the `around` advice you want to execute the operation that is at the join point, you must use a special keyword—`proceed()`—in the body of the advice. Unless you call `proceed()`, the captured join point will be bypassed. When using `proceed()`, you can pass the context collected by the advice, if any, as the arguments to the captured operation or you can pass completely different arguments. The important thing to remember is that you must pass the same number and types of arguments as collected by the advice.

Each `around` advice must declare a return value (which could be void). It is typical to declare the return type to match the return type of the join points that are being advised.

There are cases when an `around` advice applies to join points with different return types. To resolve such situations, the `around` advice may declare its return value as the common base class of these types, or even as `Object`. In those cases, if `around` returns a primitive type after it calls proceed(), the primitive type is wrapped in its corresponding wrapper type and performs the opposite, unwrapping after returning from the advice. The scheme of returning the `Object` type works even when a captured join point returns a `void` type.

#### 2.1.3.4.Introduction

The *introduction* is a static crosscutting instruction that introduces changes to the classes, interfaces, and aspects of the system. It makes static changes to the modules that do not directly affect their behavior. For example, you can add a method or field to a class. You can also modify the inheritance hierarchy of existing classes to declare a superclass and interfaces of an existing class or interface as long as it does not violate Java inheritance rules.

Most notable of these rules is, of course, the absence of multiple inheritance in Java. AspectJ provides partial workaround for this limitation. In AspectJ an aspect may introduce data members and methods with implementation into interfaces. Therefore, it is possible to create a class that inherits implementation from several ancestors – one of these ancestors may be a class, while others are interfaces with introduced implementation. This option is commonly used to provide a default behavior to the implementing classes.

The following introduction declares the `Account` class to implement the `BankingEntity` interface:

```
declare parents: Account implements BankingEntity;
```

Introduced members can be marked with an access specifier. The access rules are interpreted with respect to the aspect doing the introduction. For example, the members marked private are accessible only from the introducing aspect.

#### 2.1.3.5.Compile-time declaration

The *compile-time declaration* is a static crosscutting instruction that allows you to add compile-time warnings and errors upon detecting certain usage patterns.

For example, you can declare that it is an error to call any Abstract Window Toolkit (AWT) code from an EJB.

The following declaration causes the compiler to issue a warning if any part of the system calls the save() method in the Persistence class. Note the use of the call() pointcut to capture a method call:

```
declare warning : call(void Persistence.save(Object))
   : "Consider using Persistence.saveOptimized()";
```

#### *2.1.3.6.* Aspect

The *aspect* is a new central unit of AspectJ, in addition to class, which remains as significant as it is in Java. Aspects contain the code that expresses the weaving rules for both dynamic and static crosscutting. Pointcuts, advice, introductions, and declarations are combined in an aspect. In addition to the AspectJ elements, aspects can contain data, methods, and nested class members, just like a normal Java class. Aspects can be declared to be abstract. With abstract aspects, you can create reusable units of crosscutting by deferring some of the implementation details to the concrete subaspects. An abstract aspect can mark any pointcut or method as abstract, which allows a base aspect to implement the crosscutting logic without needing the exact details that only a system-specific aspect can provide. An abstract aspect by itself does not cause any weaving to occur; you must provide concrete subaspects to do so. An aspect that contains any abstract pointcut or method must declare itself as an abstract aspect. In this respect, aspects resemble classes. Any subaspect of an abstract aspect that does not define every abstract pointcut and method in the base aspect, or that adds additional abstract pointcuts or methods, must also declare itself abstract.

# 3. Motivation

Great deal of work has been done in the last years on combining AOP and refactoring. This work focuses mainly on finding crosscutting concerns in “conventional” non-AOP software, and extracting them into aspects [6].

My thesis focuses on manipulating code the other way around – producing “conventional” source code from aspect-oriented code. Since AOP is an efficient tool for crosscutting modification, we would like to use this tool as often as needed. In this case, however, the understandability of the source code may suffer. For example, an aspect may replace execution of a function with an execution of completely different portion of code, and nothing will indicate that this replacement has happened in the code that calls the original function. Such situation is difficult to track and hence should be avoided.

Furthermore, in some existing systems written in object-oriented languages it is impractical to use aspects in the late stages of its development, after most of the system has been designed and implemented using conventional object-oriented methods. Most companies will not even consider this option, since AOP is still considered an experimental approach and is not widely used in industry. In this case it may seem safer to use AOP only during the modification, and not to rely on its availability throughout the entire system’s lifecycle.

I have developed a new tool called ACME (acronym of Automated Crosscutting Modification in Eclipse), which solves this conflict by automatic conversion of AspectJ code to pure Java.

A typical workflow when using ACME would be as follows: when there is a need for crosscutting modification, the programmer introduces the required change in an aspect. He or she evaluates different solutions and tests the program. After the programmer is satisfied with the results, ACME is applied and produces equivalent code in pure Java. Thus, the programmer is able to enjoy the benefits of AOP during development, yet does not have to keep aspect-oriented code in the production version of the system.

## 3.1. What is a Crosscutting Concern and What Is Not

A typical crosscutting concern can be identified even in an informal description of system functionality. Someone describing such a concern will usually say that execution of any action in a group of system actions should also involve some other actions, which are not directly related to it. For example, phrases like “all function calls must be logged” or “any manipulation of bank accounts must be preceded by the authentication of the user” represent crosscutting concerns.

Let’s consider some detailed examples.

The first example is taken from [5]. It deals with user authentication in a banking system. This example introduces an abstract aspect, which defines the general behavior of an authorization module. The aspect declares an advice which is executed whenever a function that requires authorization is called, just before the execution of this function. This advice, which is actually a piece of code in Java, checks whether a user is already logged into the system. If not, the function attempts to log the user in.

This abstract aspect can be placed in a reusable library and used in any application. Anyone using this library only needs to define a concrete aspect, which will provide the exact list of functions, for which the advice of the abstract aspect must be applied.

Another example is taken from the developers of the AspectJ language. It deals with a telecom demo application, and is fully described in [7]. The goal of this example is to demonstrate complete separation of business concerns using aspects. The key concern of a telecom system is to support telephone calls between customers. This concern is implemented in the classes *Customer, Call, Connection* and their derivatives.

Another concern is billing the client. Its implementation is located in an aspect named *Billing*. Placing it in an aspect completely separates it from code implementing other concerns. This aspect contains calculations related to billing and also defines when these calculations should be executed, using pointcuts.

Of course, it is possible to implement the billing concern in a class instead of an aspect. In this case, however, it is necessary to insert calls to this class’s method in the implementation of the key concern classes. This means that the separation of concerns that a pure object-oriented solution can provide is not as good as one offered by an aspect-oriented solution.

Following is an example of an aspect that does not represent a crosscutting concern, but rather a crosscutting modification (also taken from [5]). It shows an

implementation of database-connection pooling. Here the motivation for using aspects is different from the earlier examples. The author first shows a conventional implementation of connection pooling and claims that it requires changing “each creation and destruction method of the connection object to use the pooling interface”. In fact, this means that every point where connection object is created or released must be changed. To avoid this, the author offers to use an aspect that will encapsulate the change. Introduction of this aspect limits the number of places that must be changed.

This solution, however, has its own problems. In general, it is a significant disadvantage of an aspect-based architecture that the program behavior is very hard to follow for a reader. This is caused by the fact that no explicit calls for certain portions of code (advices) appear at the points these portions are executed. The implementation described above brings this disadvantage to extreme. Not just code is executed without a traceable invocation; some statements that do appear in the program text are not executed at all.

In the next section we will elaborate this problem and introduce a solution for it and several other problematic cases.

## *3.2. Refactoring Aspects Away*

This work presents several patterns in which conventional Java code can replace aspects. There are several guidelines that these patterns should follow.

The most obvious guideline is that the behavior of the Java code produced by elimination of aspects must be exactly the same as the behavior of the original code. Furthermore, the Java code should not be significantly longer than the aspect code. No programmer would like a tool that will force him to deal with an increased number of source lines. The produced code should also allow easy modification, if needed. If new modules are added to the system later, it should be possible to use the generated Java code from these modules easily. In the optimal case this use should involve simple calls to functions in generated code, and not require modification of the generated code itself.

Finally, the generated code must look similar to code written by a human programmer as well as understandable to programmers. In practice, this means that such a code

will be based on commonly used patterns and idioms. This is a critical requirement, since our main goal is to improve the understandability of the code by replacing aspects by Java.

Please note, that although ACME has some similarity to an AspectJ compiler, its goal is completely different. Java and AspectJ compilers create bytecode that can be immediately executed by a JVM. ACME is used to create source code that will be maintained by a programmer.

Even if an AspectJ compiler uses Java source code as an intermediate output, this output is not expected to be easy to understand or maintain for a human programmer. For the same reason, decompiling tools such as JODE [11] or JAD [12] cannot be used instead of ACME. The Java code they produce will be even less readable than that generated by an AspectJ-to-Java compiler.

# 4. ACME Functionality

## *4.1. Creating a Java Class*

Consider a software system that allocates many resource objects, and the creation of these objects takes considerable time. It may be helpful to pool these objects and reuse them instead of creating a new object every time an object is needed. This would be easily achieved, if a dedicated “factory” module would manage object lifecycle. Then it would have been sufficient to rewrite this module. In this case all creation operators for the resource class would have been concentrated in the factory class, and could have been easily replaced by calls to the resource pool’s methods.

However, the creation operators are more often scattered in the code than kept in a single factory class. Thus, it is necessary to introduce this factory module and replace all the calls to the creation operators by a call to this module’s methods. Obviously, this is a time-consuming and error prone task.

An aspect oriented environment provides a simpler way to solve this problem. It enables a programmer to define a pointcut containing all the points were a resource object is created. Furthermore, this pointcut is very simple to write, because it is possible to match all calls to the constructor of the resource class with a single statement in the pointcut definition. With such a pointcut available, the programmer can define an advice on this pointcut, which will get the requested object from the pool. Thus, the goal can be achieved with very little effort.

Following is an aspect that introduces pooling mechanism into existing code.

```
public aspect ResourcePoolingAspect {
   private ResourcePool _rpool = new ResourcePool();

   pointcut resourceCreation(ResourceDescription rd)
      : call(Resource.new(ResourceDescription)) && args(rd);

   pointcut resourceDestruction(Resource r)
      : call(void Resource.free()) && target(r);

   Resource around(ResourceDescription rd): resourceCreation(rd) {
      Resource resource = _rpool.getResource(rd);
      if (resource == null) {
         resource = proceed(new ResourceDescription());
      }
      return resource;
   }

   void around(Resource r) : resourceDestruction(r) {
      if (!_rpool.putResource(r)) {
         proceed(r);
      }
   }
}
```

This implementation assumes that a resource is released, using the `free` method, after it is no longer needed. This is a typical situation. Since we are dealing with objects that use expensive resources, these objects are likely to have an explicit method for releasing the resources rather than rely on automatic garbage collection.

This example uses the `ResourcePool` class, which implements the object pool itself. The `getResource` method attempts to obtain an existing object from the pool. If there is an appropriate object in the pool, this object is removed from the pool and returned; otherwise, the method returns `null`. The `putResource` method places an object into the pool, and an object is disposed only if it cannot be placed in the pool (`putResource` fails). Thus, the aspect calls these pool functions instead of the original calls to `new` or `free`, and only if these fail, uses the original calls.

A possible implementation of the resource pool is shown below.

```
public class ResourcePool {
  List _pooledResources = new ArrayList();
  Map _resourceDescriptionMap = new HashMap();

  synchronized
  public Resource getResource(ResourceDescription rd) {
    List resourcesList = getResources(rd);
    if (resourcesList == null) {
      return null;
    }
    int size = _pooledResources.size();
    for (int i = 0; i < size; ++i){
      Resource resource = (Resource)_pooledResources.get(i);
      if (resourcesList.contains(resource)) {
        _pooledResources.remove(resource);
        return resource;
      }
    }
    return null;
  }

  synchronized
  public boolean putResource(Resource resource) {
    _pooledResources.add(resource);
    return true;
  }

  synchronized
  public void registerResource(
    Resource resource,
    ResourceDescription rd)
  {
    List resourcesList = getResources(desc);
    if (resourcesList == null) {
      resourcesList = new ArrayList();
      _resourceDescriptionMap.put(rd, resourcesList);
    }
    resourcesList.add(resource);
  }

  private List getResources(ResourceDescription rd) {
```

```
        return (List)_resourceDescriptionMap.get(rd);
      }
    }
```

While aspects provide an easy solution to the problem, the resulting code is complex and misleading, because creation operators - which remain from the original source code - are never executed in the new version of the system. The object-pooling aspect is not really a crosscutting concern. As we have seen, it could easily be implemented as a class in an object-oriented language. It was introduced only to solve a design problem in the existing software.

ACME provides both the ease of modification offered by the aspect-oriented environment and the understandability of the conventional solution. It can automatically convert this aspect into the following Java class.

```
public class ResourcePoolingManager {
  public static ResourcePoolingManager getInstance() {
    return instance;
  }

  private static ResourcePoolingManager instance =
    new ResourcePoolingManager();

  private ResourcePoolingManager() {
  }

  private ResourcePool _rpool = new ResourcePool();

  public Resource resourceCreation(ResourceDescription rd) {
    Resource resource = _rpool.getResource(rd);
    if (resource == null) {
      resource = new Resource(new ResourceDescription());
    }
    return resource;
  }

  public void resourceDestruction(Resource r) {
    if (!_rpool.putResource(r)) {
      r.free();
    }
  }
}
```

The generated class is automatically made a singleton [8] by adding the *getInstance* method and marking the class's constructor as private. This allows calling its methods from anywhere in the program and retains the global scope aspects have by their nature. Each advice in the aspect is replaced by a function. By default, ACME uses the names of the pointcuts, which are advised by the original aspect, as the names of the generated functions. The user, however, is given the opportunity to give these functions any other name. All the fields and methods defined in the aspect are retained in the new class.

Finally, ACME finds all the function calls in the system, which are influenced by the aspect and replaces them with calls to the singleton's methods. For example, the line

```
Resource r = new Resource(null);
```

is replaced with

```
Resource r =
   ResourcePoolingManager.getInstance().resourceCreation(null);
```

This process generates a well-structured and readable Java code. One may notice, however, that the implementation of the resource pool now consists of two classes: the generated ResourcePoolingManager and the original ResourcePool. The separation of concerns between the two classes is not very clear, so in some cases it may be helpful to merge them into one class. Currently ACME does not perform this merging, and the programmer has to make it manually. The Inline Method refactoring [2] can be used to simplify this task. In the future invoking this refactoring automatically on programmer's request will be added.

### *4.2. Inlining Aspect Code*

Consider a class representing a point on a plane. This class contains only two fields for point's coordinates, as shown:

```
public class Point {
   public int x;
   public int y;
}
```

Suppose it is needed to introduce polar coordinates to this class. In this case, this class can no longer allow direct assignment to the coordinate fields, since any change of Cartesian coordinates will also require calculation of new polar coordinates.

Thus, every access to the fields must be replaced by a call to an accessor method. Again, aspect–oriented languages provide an easy way to achieve the functionality. It is very simple to write an advice that will execute the required calculation where the original code references a data field. This is what such an aspect looks like.

```
public aspect PointAspect {
  pointcut getX(Point p)
    : get(int Point.x) && target(p);

  pointcut getY(Point p)
    : get(int Point.y) && target(p);

  pointcut setX(Point p)
    : set(int Point.x) && target(p);

  pointcut setY(Point p)
    : set(int Point.y) && target(p);

  before(Point p) : getX(p) {
    p.prepare();
  }

  before(Point p) : getY(p) {
    p.prepare();
  }

  after(Point p) : setX(p) {
    p.markChange();
  }

  after(Point p) : setY(p) {
    p.markChange();
  }

  public void Point.prepare() {
    // reconcile the Cartesian and polar coordinates
    // when of them changes
   }


  public void Point.markChange() {
    // indicate that one of the coordinates have changed
     // and prepare shall be called
  }

  // More introduced members, used for representation
```

```
    // polar coordinates
  }
```

Implementation of some parts of the aspect is not shown in order to keep the example short.

The methods *prepare* and *markChange* introduced into the *Point* class are used to reconcile the Cartesian and polar coordinates when one of them changes. There is a pointcut for read access of each of the two data fields and a pointcut for write access. For each read access there is an advice that ensures that the Cartesian coordinates are valid by calling `prepare`. The body of the function is not shown here, but it should include calculation of Cartesian coordinates based on polar ones. For each write access there is an advice that calls the `markChange` method. It sets a flag to indicate that the polar coordinates must be recalculated before they are retrieved.

Again, this is a fast way to make things work.

However, we do not really want to have the `Point` class effectively split into two modules. Applying ACME to this aspect prevents this split. It will automatically merge the aspect into the `Point` class. The result is shown below.

```
public class Point {
  public int x;
  public int y;

  public int getX() {
    prepare();
    return x;
  }

  public int getY() {
    prepare();
    return y;
  }

  public void setX(int newX) {
    x = newX;
    markChange();
  }

  public void setY(int newY) {
    y = newY;
    markChange();
  }

  public void prepare() {
    // Body skipped
  }

  public void markChange() {
    // Body skipped
  }

  // More members
}
```

The aspect's advices are transformed into new methods of the class, and members introduced by the aspect become regular class members.

Finally, ACME finds all references to the fields of the Point class anywhere in the system and replaces them with calls to the new methods. So, if `p` is a variable of class `Point`, a line of code such as

```
p.x += 56;
```

becomes

```
p.setX(p.getX() + 56);
```

At this stage we already have a working system with all functionality contained in Java classes and the data members of the `Point` class are no longer used anywhere besides this class's methods. These data members, however, are still public and can be used by some code that will be written in future. This is undesirable, so the programmer has to change the accessibility of these members to private or protected. This change is very simple for a human programmer to make, but it is trickier for a computer program. For example, it is likely to require moving the data members to a different location, because programmers usually keep members with the same accessibility together. These considerations are out of the scope of this thesis. However, one can consider providing it in a future version of ACME. The accessibility change should only be applied on programmer's explicit request, because it could not always be helpful. Therefore, it should be implemented as an option that a programmer may choose, but not as default behavior.

The same manipulation may also modify the behavior of a family of classes, not just a single class. Consider a system that contains a family of classes implementing the Command pattern [8]. Each class's execution method makes several calls to database operations. The base class of this hierarchy and a sample concrete class are like this:

```
public abstract class Command {
   public abstract void execute();
}

public class ConcreteCommand extends Command {
   public void execute() {
      // Do something
   }
}
```

Suppose that at some point it is decided that the execution of these commands must be enclosed in a transaction. The common solution for this task is to split the execute method of the Command class into two. The clients of the Command class call a method that is defined in the base Command class. This method opens the transaction, calls an internal execute method, and then commits the transaction. This internal

method is implemented in each concrete command class and it makes the same calls the original method have made.

Introducing this change into a large project may be difficult, since it requires modifying many source files. Aspect-oriented languages provide an easier way. In such languages it is possible to define an advice on execution of the base class's `execute` method. This advice will be called for any class derived from Command. It will open the transaction before the method is executed and commit after the method is completed. Such an aspect – written in AspectJ - is shown below.

```
public aspect TransactionAspect {

   pointcut protectedExecute(Action a)
      : call(void Action.execute()) && target(a);

   void around(Action a) : protectedExecute(a) {
      Transaction transaction = new Transaction();
      proceed(a);
      transaction.commit();
   }
}
```

ACME can transform this aspect-oriented version to the conventional one. Using the aspect definition it creates the new external function in the base class.

This function's name is identical to the pointcut name as defined by the programmer, `protectedExecute` in our example. The modified `Command` class looks as follows.

```
public abstract class Action {

   public abstract void execute();

   public void protectedExecute() {
      Transaction transaction = new Transaction();
      execute();
      transaction.commit();
   }
}
```

ACME also modifies the clients of this class family to call the new function. However, in this case, as opposed to the previous example, it may not be desirable. It is likely that the programmer would like to keep the interface of the class unchanged.

This requires changing the name of the existing functions and using the original name for the new function. For instance, in our example the `execute` function's name will be changed to `internalExecute` and the `protectedExecute` function will be called `execute`. The easiest way to achieve this is by invoking the refactoring operation Rename Method, which is available in the Eclipse IDE. In a future version of ACME this refactoring may be applied automatically, but as for now the programmer still needs to invoke it explicitly.

### *4.3. Converting Aspect Hierarchies*

So far we have only discussed processing one aspect at a time. In general, this is good enough. Even if several aspects are applied to the same classes and we need to convert these aspects to pure Java, this still can be done. Applying ACME to each aspect separately enables achieving this goal. The user should only pay attention to applying the transformations in an order that matches the precedence order defined on the aspects, if there is one.

However, AspectJ also enables splitting functionality between several aspects by means of aspect inheritance. Usually an abstract base aspect defines the common behavior of the aspects. Often this behavior is defined using template methods. In this case the base aspect implements the general part of the advices whereas parts the specific parts are deferred to the derived aspects. Abstract pointcuts are another common mechanism for splitting functionality in an aspect hierarchy. When using abstract pointcuts, the programmer defines the advice implementation in the base aspect but does not define to which join points the advices are applied. In the derived aspects the programmer provides concrete pointcut definitions that specify the join points to be used by the advice. Naturally, different derived aspects may specify different join points. Below is an artificial example of such a structure, which includes a base aspect and two derived aspects. A more realistic example is presented in detail in section 7, which presents a detailed case study.

```
public abstract aspect ParentAspect {
   public abstract pointcut exec();

   void around(Base.Resource r) : exec() && target(r) {
      foo();
      System.out.println("Replaced call");
   }

   public abstract void foo();
}
public aspect DerivedAspect extends ParentAspect {
   public pointcut exec() : call(void Base.Resource.free());

   public void foo() {
      System.out.println("Replaced call");
   }
}

public aspect AnotherDerived extends ParentAspect {
   public pointcut exec() :
      call(void Base.Resource.testFunction(String));

   public void foo() {
      System.out.println("Replaced call");
   }
}
```

Such a structure represents a new challenge for the implementation of ACME. ACME only supports creation of classes from aspect hierarchies. Some of the generated classes are made singleton, similar to the class generated from a single aspect. However, this is not required for other classes, as I will explain later.

On the other hand, it is probably useless to inline aspect hierarchies into existing classes: aspect hierarchies are rather complex constructs and forcing all their functionality into a single class is not likely to improve the code understandability.

Now consider the implementation of the conversion process. Though conversion of the aspects into classes is still relatively simple, there are several new issues to consider.

The most obvious is the introduction of inheritance. Since it splits the original aspect code to several modules, a similar separation must be retained in the generated Java

code. We can achieve this separation easily using a hierarchy of Java classes. Aspect can be converted to classes one at a time, as it was done in the earlier examples. The only information that ACME needs to pass from one aspect to another is the name of the generated classes, since the name of the base class must appear in the `inherits` clause of the derived class's definition.

It should also be noted that we must treat abstract aspects differently from the concrete ones. The AspectJ run-time environment automatically creates a single instance of a concrete aspect and ACME simulates this behavior using the Singleton pattern. On the other hand, abstract aspects are never instantiated. Therefore, ACME transforms them into abstract classes and does not add Singleton pattern implementation.

In addition, we must now consider abstract methods and pointcuts. Fortunately, they do not require special treatment in generation of new classes. Abstract methods simply can be copied from aspects to corresponding classes, since they are used similarly in both. Abstract pointcuts, on the other hand, are not copied at all, since any definition of an aspect, abstract or not, affects the join point locations only, not the new classes.

The generated classes are shown below:

```
public abstract class ParentManager {

  public void exec(Base.Resource r) {
    foo();
    System.out.println("Replaced call");
  }

  public abstract void foo();
}

public class DerivedManager extends ParentManager {
  public static DerivedManager getInstance() {
    return instance;
  }

  private static DerivedManager instance = new DerivedManager();

  private DerivedManager() {
  }

  public void foo() {
    System.out.println("Replaced call");
  }
}

public class AnotherDerivedManager extends ParentManager {
  public static AnotherDerivedManager getInstance() {
    return instance;
  }

  private static AnotherDerivedManager instance =
    new AnotherDerivedManager();

  private AnotherDerivedManager() {
  }

  public void foo() {
    System.out.println("Replaced call");
  }
}
```

As we have seen, supporting aspect hierarchies requires only minor changes in the process of generation of new classes. Modifying the join points is, however, significantly more complicated than before, for two reasons.

First, it is more difficult to analyze the pointcuts that specify the involved join points. Since abstract pointcuts can be used, the definition of a pointcut may be spread over several source files. Therefore, ACME must process all these files to collect the information needed to transform each aspect.

Second, in some Java files one may encounter join points that match pointcuts from different aspects. In order to correctly make all the required modifications, ACME must first create a list of all join points modified by any of the aspects and only then modify the files.

# 5. Design Decisions

The most important decision is, naturally, to define what Java code should be inserted instead of the aspects, and where it should be placed.

## *5.1. Call vs. Execution pointcuts*

At first it may seem that `call` and `execution` pointcuts only differ in the context that is available to the advice. For example, `this` keyword when used with the `call` pointcut provides access to the calling object, while the same keyword when used with `execution` pointcut references the called object. There is even more significant difference between the two types of pointcuts when inheritance is involved. A `call` pointcut selects join points according to the static type of a reference. This means that an advice on a function of a derived class will not be called when the function is invoked through a reference to a base class. On the other hand, `execution` pointcuts select join points according to the dynamic type, which means the pointcut on a function will be called whenever this function is called.

Consider, for example, base class `Base` and class `Derived`, which is derived from `Base`, and a function call

```
b.foo()
```

where `b` is defined as type `Base`. A pointcut, which is defined as

```
call(void Derived.foo())
```

will never match this call, while a pointcut defined as

```
execution(void Derived.foo())
```

will match the call in cases where `b` references an object of type `Derived`.

Hence, ACME must treat `call` and `execution` pointcuts differently. The semantic difference between `call` and `execution` pointcuts is discussed in depth in [15]

## *5.2. Call Pointcuts*

There are two separate issues that must be dealt with while converting advices to Java functions. The first is where to place the calls to the new functions. The second is the implementation of these functions.

To mimic the above-mentioned behavior, ACME must place the code it inserts instead of a `call` advice at the locations of all calls made to the effected function. Since those locations are chosen according to the static type of objects, it is easy to find them by searching the source code. There are two possible strategies of modification at the call location. We can either replace the original call with a call to a new function that is generated by ACME, or we can keep the original call and place an additional call next to it. The latter approach is problematic for two reasons. First, it makes the client code longer, because it inserts an additional call for every call to the original function. As we have already discussed, this is undesirable. Second, and more important, this approach cannot be used for all possible advices. For example, it cannot be used when an advice should be called instead of the original call, and not in addition to it. The first approach does not suffer from such problems; therefore, it is always used for processing `call` pointcuts.

The most simple case of implementation of an advice is the implementation of an advice on an `around` pointcut, that does not contain a `proceed` command. In this case the original function should not be called at all, so the new one will only include the advice code. In all other cases the new function must include both the original functionality and the functionality of the advice. As it has been described, a typical use of ACME is to create a singleton class instead of an aspect. Since the generated function will be part of this class, it cannot directly include the functionality of the original function. Instead, it is better to make this new function call the original one.

There is still a considerable problem in this implementation. A direct call to the original function may only be used if the pointcut references a single function. Otherwise, the implementation must call the correct function for each modified join point. This is not a simple task. The AspectJ compiler uses special “closure” objects, which encapsulate such calls [9]. Practically, this means that a new class is generated for each join point or, at least, for each function used in a join point. This solution works well for a compiler; however, it is not useful for a source code-generating tool like ACME. One of the key goals of ACME is to generate code that is easy to understand and maintain, and the structure described above hardly fulfils this goal. Due to this problem, ACME only supports pointcuts referencing multiple functions when they are used in an `around` advice with no `proceed` command.

In addition to the functionality that appears explicitly in advice and function bodies, ACME must also deal with exception handling. The generated code must behave similarly to the AspectJ run-time when advised functions throw exceptions. The required action differs for the various advice types. A `Before` advice does not require special treatment, because the function is, obviously, executed after the advice, and exceptions it throws cannot influence the execution of the advice. This is true for an `around` advice as well, because for this type of advice it is the responsibility of the advice itself to deal with exception. However, ACME must deal with exceptions when generating functions for `after` advices. In fact, AspectJ allows the users to define advices that match functions exiting in three different ways, and ACME must support all three. The possibilities are to execute the advice when a function returns normally, when a function throws an exception or to execute the advice in both cases. To support the first case no special treatment is needed; it is sufficient just to place a call to the original function in the generated one. For the remaining cases the call to the original function must be enclosed in a `try` statement. If the advice should be called when the function throws an exception, the advice body is placed in the `catch` part of the statement; otherwise it is placed in the `finally` part.

This formal transformation may produce functions that are unnecessarily complicated. For instance, in the `Point` class example ACME generates `try` statement around an assignment operator, which cannot throw exceptions. It is possible to process field access pointcuts differently from function call pointcuts. This different processing will not include any processing for exceptions. Practically this means that the `try` block would not be generated. However, such a feature does not provide a significant improvement. The user can achieve the same result easily by using an advice that selects only normally returning join points, instead of join points that either return normally or throw exceptions. Furthermore, using such an advice is considered a good practice in common AspectJ programming.

## *5.3. Execution pointcuts*

As I have already mentioned, execution pointcuts require selection of join points based on the dynamic type of the object whose functions are used in the pointcut. Practically, this means that the advice code must be inserted in this object's class. It can be inserted either in the same function or in a new function that will be generated by ACME. Generating a new function is problematic. This function must be declared

in all parents of the original class, including interfaces. It is very likely that such a function will have no reasonable meaning in these interfaces. So, this solution must be avoided. Thus, the original function has to be changed.

Consider now the changes that should be done in this function. In case of an `around` advice with no `proceed` command the changes are very simple: the body of the function should be replaced with the advice's body. If this function is overridden in some derived class, the overriding function must be deleted. This can still be achieved easily.

For any other advice, the function will be responsible for execution of the advice body as well as the execution of its own original functionality. Obviously, for the sake of clarity we must still keep the advice implementation visually separated from the original function's implementation. This means that in order to achieve this separation at least one new function must be introduced. For example, if foo is the original function and adviceFunction is the function generated from an advice, after ACME is applied the foo function will become

```
void foo() {
  adviceFunction();
  // original implementation of foo
}
```

for a before advice or

```
void foo() {
  try {
    // original implementation of foo
  } finally {
    adviceFunction();
  }
}
```

for an after advice which is executed both when the function returns normally and when it throws exceptions.

Next, one must decide where this function will be created and what it will look like. In general, ACME can place new functions either in the same class the original function is located, or in a new class, which is created as a replacement for the aspect. However, with `execution` pointcuts the second option has little practical value. Using an aspect or a separate class is usually intended to keep distinct concerns in separate modules. In the original structure, which contains aspects, the source code of the

aspect is dependant on the advised class's code. The aspect obviously references this class in its pointcut definition and probably contains additional references, for example, as advice parameters. Now ACME has to insert a call to the function which is replacing the aspect's advice in the new class to the original function. This call creates a dependency in the opposite direction, completing a dependency cycle. For example, if the original class is named C and the class replacing an aspect is named AspectSubstitute, the result may look as follows:

```
class C {
  …
  void foo() {
    AspectSubstitute.getInstance().adviceFunction(this);
    …
  }
  …
}
class AspectSubstitute {
  …
  public void adviceFunction(C target) {
    …
  }
  …
}
```

This cyclic structure violates the modularity principle of decomposability [14], and should be avoided.

So, it turns out that the modification must create a new method in the existing class. The simplest way to achieve the needed functionality would be to place the advice body in the original function, and move this function's body to a new function. In addition, it would be necessary to rename all overrides of the original function in derived classes. Since the advice affects them as well as the original function, in the modified structure only the new function should be overridden, and the names must be changed to achieve this.

However, while this modification is theoretically possible, its practical application is very limited. If the advised function does not override a function of some base class, the behavior of an `execution` pointcut does not differ from the behavior of a `call` pointcut, which can be processed by ACME. On the other hand, if this function is never overridden in derived classes, the modification only changes a single location in

the source code, and this hardly justifies introduction of an automatic tool. The remaining cases are rather rare, and therefore the practical benefit of this modification is not significant. For this reason it is currently not implemented.

# 6. AspectJ Language Support

ACME is designed to perform refactoring transformation whenever the AspectJ source code follows one of several known patterns. Therefore, it only needs to support a subset of the AspectJ language, which is used in those patterns. One may consider, however, supporting additional patterns in a similar manner. In order to identify the cases when such an extension would be possible, one must first identify language constructs that allow it. As explained in section 3.2, there are significant rules that the produced Java code must follow. Obviously, some AspectJ constructs cannot be converted to such Java code. Claiming the opposite would practically mean that the AspectJ language is not needed at all. However, the exact list of these constructs is, to some extent, a matter of personal choice. Some programmers may use certain language structures freely, while other will normally avoid them. In this section I describe several AspectJ constructs that cannot be supported by ACME.

## *6.1. Wildcards*

Wildcards are an important feature of the aspect-oriented architecture. They allow defining a rule for selecting join points while not making an implicit list. This is an extremely flexible mechanism. A pointcut, which uses wildcards, may match functions that are written after the pointcut has been defined.

This flexibility practically disallows converting advises that use wildcards into functions. The automatic conversion made by ACME will modify calls to all the existing function as needed. However, it removes the aspect from the system. Therefore, if a new function is added, the aspect will not be applied to it. This means that the conversion modifies system’s behavior, and must be avoided.

A possibility that may be considered is to keep both the version with aspects and the aspect-free version of the system available. This option may allow running ACME again when new join points are added that match the pointcuts. It is probably useful when the use of ACME is due to the need to avoid using aspects in production code.

It must also be mentioned that even if we try to implement this possibility, the problems described in Section 6.3 still make the conversion impractical in many cases.

## 6.2. Type-based Matching

AspectJ pointcuts may select join points according to the type of objects involved. For example, consider the classes `Base` and `Derived` where `Derived` is subclass of `Base` and function `foo` declared in `Base`. An advice defined with the pointcut

```
call(Base.foo()) && target(Derived d)
```

is only invoked when the actual type of the target object is of type `Derived` or its subclass. The generated Java code must mimic this behavior. As we have already discussed, the generated code for the **Java Class Creation** is not placed in the original classes. This means that polymorphism cannot be used to distinguish objects of the `Derived` class from objects of `Base` class or its other subclasses.

The pointcut may also check the type of arguments passed to the function. In this case polymorphism cannot be used either for **Java Class Creation** or **Inlining Aspect Code.**

Therefore, explicit type casting must be used. The function that replaces the advice checks whether its parameter is of the requested type, and call the advice code, if so; otherwise, it will call the original function. This an example of such a function:

```
void adviceFunction(Base b) {
    if (b instanceof Derived) {
        Derived d = (Derived) b;
        // Execute advice logic
    } else {
        b.foo();
    }
}
```

This function structure is usually considered poor coding style by itself. It becomes much worse when there are several pointcuts on the same function, which differ on the target object's type. Such pointcuts would have been transformed into several if blocks.

One may attempt to use the `Visitor` pattern [8] to avoid this structure. This pattern is often used when a multiple dispatch structure is required, as in this case. However, using this pattern will require generation of rather complicated and long code. In addition, this solution cannot be scaled to support several pointcuts in an aspect. Each pointcut must be supported by its own set of `advice` and `visit` functions, which

means functions with awkward names such as `visitForAdviceA` will have to be created.

These attempts to find equivalent Java code for the aspect show that any solution has considerable drawbacks. In order to discourage creating any of the described code structures, ACME does not support aspects that use type-based matching. ACME can still, however, be helpful in converting such an aspect to Java code. A programmer may rewrite the advice to use explicit type casting or `Visitor` pattern instead of the type-matching pointcut. To achieve this he only needs to modify the aspect itself. Afterwards, he will be able to run ACME to replace the aspect with a singleton class and update the effected calls. This limits the manual modification to one source file or a small number of files.

### *6.3. Implicit Type Casting*

The AspectJ language allows the types of advice parameters and the return values to differ from the corresponding types in the advised functions. It is only needed to consider the cases when the type in an advice is either a subclass or a superclass of the type in the original function. Any other case either behaves in a similar way, or yields no join points at all.

When an advice parameter's type is a superclass of the function parameter's type the transformations introduced in Section 4 will produce valid Java code. In the client code, ACME replaces an existing function call by a call to the function it generates from an advice. The new call expects to receive as an argument an object of a type that is more general than the type expected by the original call. Obviously, the object that the client has been using can be passed safely. In fact, this situation is similar to contravariant redefinition of function parameters. [14]

In a similar manner, ACME can correctly process the case when an advice return type is a subclass of the original function's return type. This case resembles covariant redefinition of the return type, which is known to be type-safe.

We have already analyzed what happens when the advice parameter's type is a subclass of the original parameter's type in the section on type-based matching. The only remaining case is when the advice's return type is a superclass of the original function's return type, as in the following advice definition:

```
Object around() : call(String B.foo()) {…}
```

In this case, in order to achieve Java code that produces the same results as the aspect code, an explicit type cast must be used. The straightforward solution is to place this type cast in the client code at every location that ACME modifies. Theoretically, this solution should not be considered well structured, since it makes heavy use of type casting. However, until recently, most Java code has suffered from this problem due to the absence of generics from the language. The recent introduction of generics in Java version 1.5 may improve the style of general Java code, but will also allow for a better solution for our problem. Unfortunately, the currently available version of the AspectJ environment (1.2) does not support generics, although the next version (5) is expected to fully support this important feature.

The need for the advice to use a type, which is more general than the type used by the original function, usually arises when the same advice is applied to different functions. Often, this is achieved using aspect hierarchies. The base aspect defines the advice and an abstract pointcut. The derived aspects define concrete pointcuts for the advice and, possibly, some protected functions that are used by the advice. These concrete pointcuts may refer to functions with different return types; therefore the abstract pointcut and the advice have to use the common parent of these types. Example from Section 4.3 can be modified to show this situation. All that is needed is to change the return type of the advice and of the functions referenced in the pointcuts.

```
public abstract aspect ParentAspect {
  public abstract pointcut exec();

  Object around(Base.Resource r) : exec() && target(r) {
    System.out.println("Replaced call");
    return foo();
  }

  public abstract Object foo();
}
public aspect DerivedAspect extends ParentAspect {
  public pointcut exec() : call(Integer Base.Resource.foo());

  public Object foo() {
    return new Integer(1);
  }
}

public aspect AnotherDerived extends ParentAspect {
  public pointcut exec() :
    call(String Base.Resource.testFunction(String));

  public Object foo() {
    return “Test Function”;
  }
}
```

In AspectJ version 5 the same functionality may be achieved by using the following aspects, which employ generics:

```
public abstract aspect ParentAspect<C> {
  public abstract pointcut exec();

  C around(Base.Resource r) : exec() && target(r) {
    System.out.println("Replaced call");
    return foo();
  }

  public abstract C foo();
}

public aspect DerivedAspect extends ParentAspect<Integer> {
  public pointcut exec() : call(Integer Base.Resource.foo());

  public Integer foo() {
    return new Integer(1);
  }
}

public aspect AnotherDerived extends ParentAspect<String> {
  public pointcut exec() :
    call(String Base.Resource.testFunction(String));

  public String foo() {
    return “Test Function”;
  }
}
```

It is possible to convert these aspects into Java classes in a process similar to that shown in Section 4.3. The results of these conversions are shown below:

```
public abstract class ParentManager<C> {

  public C exec(Base.Resource r) {
    System.out.println("Replaced call");
    return foo();
  }

  public abstract C foo();
}

public class DerivedManager extends ParentManager<Integer> {
  public static DerivedManager getInstance() {
    return instance;
  }

  private static DerivedManager instance = new DerivedManager();

  private DerivedManager() {
  }

  public Integer foo() {
    return new Integer(1);
  }
}

public class AnotherDerivedManager extends ParentManager<String> {
  public static AnotherDerivedManager getInstance() {
    return instance;
  }

  private static AnotherDerivedManager instance =
    new AnotherDerivedManager();

  private AnotherDerivedManager() {
  }

  public String foo() {
    return “Test Function”;
  }
}
```

This conversion is currently not supported by ACME yet, because the required AspectJ version is not released yet.

### 6.4. Control Flow Pointcuts

These pointcuts select join points that are encountered during the execution of a certain function. There is no structure in common object-oriented languages that can be used to implement similar behavior. One can attempt to use a flag variable to indicate that the program execution enters the function and test this flag when the join point is reached. The resulting code, however, does not provide a clear separation of concerns and is very hard to deal with for a human programmer, because it is split to several small portions located in different places and even in different files.
Hence, supporting this language feature in ACME will not provide any benefit.

### 6.5. Compile-Time Aspects

These are aspects that modify the behavior of the compiler as opposed to the behavior of the program in which they appear. For example an aspect may declare that no join point may exist that matches certain pointcuts. There is no similar structure in Java; therefore, it is impossible to support compile-time aspects in ACME.

# 7. A Case Study

## *7.1. NMS Introduction*

The work by S.Raz [10] is a typical example that clearly demonstrates the need for ACME. This work discusses scaling up NMS, a network management system. The original version of the system was designed for relatively small networks, so it was keeping the entire network configuration in memory and only required loading data from a database on system startup. After the need had appeared to support larger networks, this strategy became inadequate.

The preferred strategy for large networks is based on keeping the network configuration in a database and loading data when needed. The author describes a process introducing this strategy. This process involves a series of code modifications; each modification involves replacing an implementation of a certain data access operation. NMS uses two types of such operations. One is retrieving a single object, such as device or port from the database, using the object ID. The other is retrieving objects that are related to a given object. Examples of this type of operations are retrieving the list of all ports of a hardware device or retrieving the device that owns the specified port.

The work shows that these modifications can be done using AOP. Furthermore, aspects allow introducing the changes quickly and easily. They also allow a programmer to experiment with different solutions, evaluate their behavior and choose the best one. The use of aspects is especially important because this reengineering effort is undertaken while the system is continuously developed and other, unrelated features are introduced into the system. Aspects provide the complete independence of the unrelated efforts.

This aspect-oriented approach was tested on a demonstration system, which was similar to the production system. However, while aspects are helpful in testing the changes, the possibility of using aspects for evolution of a production system seems to be unlikely. The major limitation is that it is practically impossible to introduce such an innovative tool into large commercial system, especially in a relatively late stage in its lifecycle.

This is the situation, in which ACME may be especially helpful. In this section I discuss the applicability of ACME to the aspects shown in Raz’s work His work

shows both development and production aspects. Development aspects are aspects that are used during the development of a system only. For example, they may be used to collect profiling information about system execution. Obviously, it is never needed to convert development aspects into conventional code, since they are not included in the production code anyway. Therefore, it is only needed to consider production aspects.

One must notice that the aspects were written with no automatic conversion tool in mind. Therefore, it is not surprising that ACME cannot process them as they have been written originally. It is good enough to show that ACME can process aspects that have the same effect and are not more difficult to write. In fact, the original aspects use `execution` pointcuts, which cannot be processed. However, in most of these cases no polymorphic calls are involved, and the behavior of `execution` pointcuts does not differ from that of `call` pointcuts. This means that practically the same pointcuts may be processed.

## 7.2. Retrieving List of Related Objects

The simplest operation in Raz's thesis is retrieving the list of related objects. Its implementation is split to several aspects. The *AbstractGetRelatedIdsAspect* is an abstract aspect that defines behavior, which is common to loading lists of all types of objects. It is shown below:

```
public abstract aspect AbstractGetRelatedIdsAspect {

  public abstract pointcut execOfGetRelatedElements(
    Element owner);

  protected abstract String getSelectString(int ownerID);
  protected abstract Class getRelatedClass();

  Collection around(Element owner) :
    execOfGetRelatedElements(owner) {
    int ownerID=owner.getId().intValue();
    return getRelatedElements(ownerID);
  }

  private Collection getRelatedElements(int ownerID) {
    Collection relatedElements=new LinkedList();
    Class relatedClass = getRelatedClass();
    Collection ids=loadRelatedIds(ownerID);
    Iterator idsIt=ids.iterator();
    while (idsIt.hasNext()){
      Integer id=(Integer)idsIt.next();
      if (id!=null && id.intValue()!=0) {
        Element element=(Element)ServerImpl.getInstance().
          getElement(relatedClass,id);
        relatedElements.add(element);
      }
    }
    return relatedElements;
  }

  private Collection loadRelatedIds(int ownerID) {
    Collection relatedIds=new HashSet();
    String select = getSelectString(ownerID);
    try {
      Connection conn =
        DbConnectionManager.getInstance().getConnection();
      Statement state = conn.createStatement();
      ResultSet rs = state.executeQuery(select);
      while (rs.next()) {
        Integer id=new Integer(rs.getInt(1));
        relatedIds.add(id);
```

```
        }
      } catch (SQLException e) {
        e.printStackTrace();
      }
      return relatedIds;
    }
  }
```

In addition, there are several concrete aspects that are derived from *AbstractGetRelatedIdsAspect*. One of them is DeviceGetRelatedPortIdsAspect, which retrieved all ports owned by a specific device.

```
public aspect DeviceGetRelatedPortIdsAspect
  extends AbstractGetRelatedIdsAspect {

  public pointcut execOfGetRelatedElements(Element owner) :
    execution (public Collection Device.getPorts()) &&
    this(owner);

  protected String getSelectString(int ownerID) {
    return "select port_id from port where device_id=" + ownerID;
  }
  protected Class getRelatedClass(){
    return Port.class;
  }
}
```

This structure is very similar to the structure shown in Section 4.3. The only difference from that structure is the use of `execution` pointcut instead of a `call` pointcut. However, in this case, a `call` pointcut can be used at the same place with no change in program behavior. After the pointcut definition had been changed, ACME successfully converted these aspects to classes.

## 7.3. Retrieving Single Related Object

Retrieving a single related object is trickier. Its implementation is very similar to the implementation shown above; however, there are some subtle differences. Since this aspect deals with retrieving single objects, its pointcut matches functions that return objects of specific types and not general collections. This is a typical pointcut used for retrieving a single object:

```
public pointcut execOfGetRelatedElement(Element owner) :
   execution (public Device PortImpl.getDevice()) &&
   this(owner);
```

Due to this change, the definition of the advice also changes; now the advice's return type must be declared as `Object`, because it replaces several functions with different return types.

These changes make the structure similar to one shown in section 6.3. Currently ACME cannot process this structure; however, introduction of generics into Java language may make it possible. On the other hand, it is possible to rewrite the aspects in a way that avoids using such a structure. For example, one may implement the same functionality without using abstract pointcuts in a base aspect. In this case it is possible to use the return type of the original function (`Device` in our example) as the advice's return type as well. After making this minor modification, one will be able to convert these aspects to Java using ACME.

## *7.4. Retrieving Objects by ID*

In the original system all data objects are held in memory data structures. There are several types of these objects, such as `Device`, `Port`, etc. All objects of the same type are held in a special manager object. For each manager object a special class is defined; these classes share a common base class, named `ElementManagerImpl`.

Retrieving the objects from the database is split into several aspects. The base aspect, `AbstractGetElementAspect`, defines most of the functionality needed for the retrieval, while the child aspects define the parts specific for each data object type. The advice used by all these aspects is defined in the base aspect and it is always applied to the same function, `ElementManagerImpl.getElement(Integer)`. During program execution, the appropriate aspect is selected according to the type of the manager object on which the `getElement` function is called in each case. This selection is made by means of a `target` pointcut.

The base aspect and an example child aspect are shown:

```
public abstract aspect AbstractGetElementAspect {

  public abstract pointcut managerTarget();

  public pointcut callOfGetElement(Integer id):
    execution(public Element
      ElementManagerImpl.getElement(Integer))
    && args(id)
    && managerTarget();

  public abstract String getSelectString(Integer id);
  public abstract Element
    createElement(ResultSet rs) throws SQLException;

  Element around(Integer id) : callOfGetElement(id) {
    return loadElement(id);
  }

  private Element loadElement(Integer id) {
    Element element = null;
    String select = getSelectString(id);
    try {
      Connection conn =
        DbConnectionManager.getInstance().getConnection();
      Statement state = conn.createStatement();
      ResultSet rs = state.executeQuery(select);
      if (rs.next()) {
        element = createElement(rs);
      }
    } catch (SQLException e) {
      e.printStackTrace();
    }
    return element;
  }
}
```

This aspect is used to load devices:

```
public aspect DeviceGetElementAspect
   extends AbstractGetElementAspect {

   public pointcut managerTarget() : target(DeviceManagerImpl);

   public String getSelectString(Integer id) {
      return "select * from device where device_id=" + id;
   }
   public Element createElement(ResultSet rs) throws SQLException {
      return DeviceDbManager.createDevice(rs);
   }
}
```

ACME cannot convert aspects that use `target` pointcuts to select join points. Furthermore, the given implementation of the aspects implies that selection of a program portion to execute is based on the run-time type of an object. Therefore, any attempt to reproduce similar behavior in pure Java code must involve modification of the code of the manager classes, as explained in section 7.4. ACME doesn't support such a modification for any AspectJ construct. Fortunately, in this case it is relatively simple to make the required change without using any automatic tool.

## *7.5. Disabling data preloading*

One of the problems of the original NMS system is the long time that is used on system startup to load all data objects to memory. Since the aspects load this objects at the point they are required, loading the objects on startup is no longer needed. The following aspect prevents its invocation.

```
public aspect PreventLoadAspect {
   /**
    * pointcut of loading elements
    */
   public pointcut execManagersElementsOperations()
      : execution(public void ElementDbManager+.load())
      || execution(public void ElementManagerImpl+.addElement(..))
      || execution(public * ElementManagerImpl+.removeElement(..));

   public pointcut execSetRelations()
      : execution (
        static private void ElementDbManager+.setRelations(..));
   /**
    * skip loading of elements
    */
   Object around ()
      : execManagersElementsOperations() ||execSetRelations(){
      return null;
   }
}
```

This aspect cannot be converted as is, since it uses wildcards and `execution` pointcuts. However, a programmer can safely and easily replace both wildcards and `execution` pointcuts by other structures. Each part of the `execManagersElementsOperations` pointcut references several overrides of a single function defined in a base class. For some of these parts the base class is `ElementDbManager`, while for others it is `ElementManager`. This means that the wildcards have practically no effect. For example, the expression

```
public void ElementManagerImpl+.addElement(..)
```

references the same functions as

```
public void ElementManager.addElement(Element element)
```

For the same reason, there is no significant difference between `execution` and `call` keywords in this case. Since all overrides match the original pointcut, the same functions will match a `call` pointcut as well.

The `execSetRelations` pointcut references static functions; therefore in this case the programmer may replace the `execution` keyword with `call` as well. Furthermore, in the example system there is only one function that matches this pointcut, so the

wildcards in the pointcut definition may be replaced by an exact function declaration. It is possible that in a real system there will be several matching functions, but it is likely that there number will be small enough, so they can be explicitly listed in the pointcut definition.

The advice in this aspect must also be changed. In its present form its return type is defined as `Object`. This is needed in order to use the advice with function that return a value as well as functions that are declared as `void`. There is no similar structure in Java; therefore we must create two separate advices, one for each return type, before invoking the conversion.

Now we can generate a Java structure that replaces this aspect. However, it is not likely that any programmer will ever want to keep such a structure in a program, because it consists of a singleton class that contains two functions that do nothing and is, therefore, absolutely unnecessary. A possible solution is to use Inline Method refactoring to remove all references to this functions and then remove the singleton class from the project.

# 8. Future Work and Conclusions

The agile development approach welcomes changing requirements even late in the development process. Refactoring is a major technique used to cope with changes. There are commercial tools that support refactoring in programs written in Java and other object-oriented languages. Research is also in progress to introduce aspects into object-oriented programs automatically. ACME provides the programmer an opposite transformation. It refactors aspects out of a program, generating a pure object-oriented structure.

This new capability can be used in numerous different situations. Most obviously, it can be used when a structural change, which makes an aspect inappropriate, is required. In addition, ACME allows using aspects in some cases, in which aspect usage has been avoided earlier. For example, in an organization that prohibits using aspects in production code, it is now possible to use aspects during development, and convert them to object-oriented code when required. ACME also provides an opportunity to experiment with possible implementation improvements, such as introduction of resource pooling, without modifying existing classes. When an improvement is shown to be helpful it may be incorporated in the system automatically.

As we have seen, the goal of ACME is to convert AspectJ code into Java code. But one can view the work on ACME in a broader context, and view it as a contribution to refactoring in general. The basic motivation of refactoring is to allow the programmer to evolve the code and obtain the most suitable design for the functionality that is already written in. In this context, the transformations developed for ACME may be used even if they do not remove all the aspects present in the code. ACME transformations are then used along with other refactoring transformations, such as those that deal strictly with java code, as well as transformations that attempt to introduce aspects to replace scattered and tangled code that implements crosscutting concerns.

For example, ACME may be helpful when it is required to add some code to a system in such a way that existing aspects will not affect this code. In this case one may apply ACME in order to convert the aspects to Java code and then introduce the new code.

Aspects will not affect the new code, because they will have been removed from the system by the time of creation of this code.

This possibility is especially important when a system is built by combining several unrelated parts, and all parts contain aspects. It may happen that in the combined system the aspects will affect the parts that these aspects were not intended for. The resulting system may have unpredictable and probably undesirable behavior. Using ACME to remove some of the aspects before combining the system parts will solve this problem.

The current version of ACME is based on examples of potentially useful patterns of conversion from aspect-oriented to conventional code, such as creation of a singleton class from an aspect. Naturally, a possible development would be to find new such patterns and implement them.

However, applicability of ACME has some limitations. The tool cannot support the entire AspectJ language. For some language constructs, such as complex wildcard expressions or declarations of compile-time errors, this limitation is natural. On the other hand, AspectJ is a very flexible language, and some tasks can be solved in several ways, using different language constructs. In some cases one of this constructs can be processed by ACME, while others cannot, because they represent behavior that generally cannot be implemented in an object-oriented language. The distinction in this usage may be subtle and it is best to consider it while creating the aspects. Therefore, the easiest use of ACME is to incorporate aspects that are created as experimental modules.

On the other hand, AspectJ language constructs preventing ACME execution in some situations are equivalent to other constructs, which are supported by ACME. To help the user in these situations ACME may offer to treat the unsupported in the same manner as the supported ones. For example, it may offer to treat execution pointcuts as **call** poincuts. It is important to notice that ACME cannot decide on its own if this treatment is valid, therefore it can only offer this change, but not perform it automatically.

In addition, some extensions that are not directly related to aspects are also possible. For instance, in some of the examples we have seen that after ACME is applied to the system, some additional modifications may be required, such as inlining the generated methods or changing accessibility of some members. These modifications may also be performed automatically as a part of ACME functionality. However, since such

modifications are an addition to ACME's core functionality, they are not applicable in all cases when ACME is used. Therefore, in each case a programmer must have the opportunity to decide if he wants to apply the additional modifications.

The case study shows that it is possible to convert aspects that have been written without intended use of ACME in mind, but some preparation for conversion have been required.

# 9. References


[1] K. Beck et al. *The Agile Manifesto*. Website, 2001. Available at http://agilemanifesto.org

[2] M. Fowler et al. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999

[3] http://www.eclipse.org

[4] http://www.jetbrains.com/resharper/

[5] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003

[6] R. Laddad. *Aspect-Oriented Refactoring.* Available at http://www.theserverside.com/articles/article.tss?l=AspectOrientedRefactoringPart1

[7] *Production Aspects – Examples* http://dev.eclipse.org/viewcvs/indextech.cgi/aspectj-home/doc/progguide/examples-production.html?rev=1.2

[8] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[9] E. Hilsdale, J. Hugunin. *Advice Weaving in AspectJ*, Proceedings of the 3rd international conference on Aspect-oriented software development. Available at http://portal.acm.org/citation.cfm?id=976276

[10] S. Raz. *A Set of Tools to Solve NMS Scaling Using Aspects*. MSc Thesis, Tel Aviv University 2005

[11] http://jode.sourceforge.net

[12] http://www.kpdus.com/jad.html

[13] S. Rura. *Refactoring Aspect-Oriented Software*. BA thesis, Williams College, 2003

[14] B. Meyer. *Object Oriented Software Construction*, Prentice Hall, 1997

[15] O. Barzilay, Y. Feldman, S. Tyszberowicz and A. Yehudai, *Call and Execution Semantics in AspectJ*, In C. Clifton, R. Lammel, and G. T. Leavens, editors, Languages Workshop at AOSD 2004