Technical Report SOCS-TR-2004.7, November 2004

Comparing Different AOP Approaches

Yang Yu
McGill University
3480 University
Montreal, QC H3A 2A7, Canada
Yang.Yu@mail.mcgill.ca

Albert Bachand
McGill University
3480 University
Montreal, QC H3A 2A7, Canada
Albert.Bachand@mail.mcgill.ca

Jörg Kienzle
McGill University
3480 University
Montreal, QC H3A 2A7, Canada
Joerg.Kienzle@mcgill.ca

ABSTRACT

Over the last few years, different aspect-oriented programming approaches have been developed, all of which can be used by a developer to modularize crosscutting application concerns at the code level. Each approach, however, is based on different technology, provides different abstractions, mechanisms, modules and language extensions, and ultimately leads to a different application design. This paper describes an experiment, in which the same application has been implemented using three different AOP approaches: AspectJ, Hyper/J and AspectWerkz. The differences between the three designs are highlighted, and the technology, language features, ease-of-use and reuse capabilities of the approaches are compared.

Keywords

Concerns, Aspects, Design Space, Reuse, AspectJ, HyperJ, AspectWerkz.

1. INTRODUCTION

In a traditional object-oriented programming approach, source code is structured using classes, which are designed by decomposing the problem into a set of self-contained abstractions based on the main functionality of the application. As a result, code implementing secondary concerns is often *scattered* throughout the source modules, and likewise code implementing different concerns ends up *tangled* inside a single module. This phenomenon is known as the "tyranny of the dominant decomposition" [1].

Aspect-oriented programming (AOP) [2] is a new programming approach that allows developers to partition their code based on the application concern it implements. Ultimately, the programmer relies on the underlying AOP environment to *weave* (or compose) the concerns together into a coherent program. Separating the expression of multiple concerns in programming systems promises increased readability, simpler structure, adaptability, customizability and better reuse.

Over the last few years, different aspect-oriented programming approaches have been developed, all of which can be used by a developer to modularize certain application concerns at the code level. Each approach, however, is based on different technology, provides different abstractions, mechanisms, modules and language extensions, and ultimately leads to a different application design.

In order to perform a comparison between different AOP approaches, a simple telecom application has been implemented using three aspect-oriented extensions of Java: AspectJ, HyperJ, and AspectWerkz. The telecom application has been chosen because of its simplicity, and because it is composed of simple, but yet still interesting concerns: establishing and handling of calls, timing calls, billing customers, and persistence of customer data. The three AOP environments have been chosen because they are all extensions of the Java language, and because of the stability of their implementation.

The paper is structured as follows. Section 2 gives an overview of the three aspect-oriented programming environments used for the experiment, and compares the weaving technology they use and the language features they offer. Section 3 introduces the telecom application, and presents the design of the most important part of the implementation, namely the one dealing with establishing and handling calls. Section 4 presents the timing aspect, and its corresponding implementations. Section 5 highlights the issues when implementing billing, and Section 6 addresses persistence. Section 7 presents a comparison of the approaches, focusing on ease-of-use and reuse potential. Section 8 mentions related work, and finally Section 8 presents the conclusion and future work.

2. ASPECT-ORIENTED PROGRAMMING APPROACHES

Over the last few years, different aspect-oriented programming approaches have been developed, providing different abstractions, mechanisms, modules and language extensions, and using different implementation technology.

One of the main elements of an AOP language is the join point model. It describes the "hooks" where concern implementations can be woven together, and thus has a big influence on the design of the application. AOP languages are supposed to provide means to identify join points, specify behavior at join points, define modules that group together behavior and structural components that implement a certain concern, and finally provide means for assembling such units to form a program. The following subsections introduce the three approaches that have been used in this experiment, followed by a comparison of the technology used and the language features they offer.

2.1 AspectJ

AspectJ [3] is the most advanced and mature AOP environment currently available. It was initially developed by a team of

programmers at Xerox Parc, and is now a continuing Eclipse project. The version used for our experiments is version 1.2.

AspectJ takes an *asymmetrical*, or "extension-like" approach to aspect-oriented programming: when designing an application using AspectJ, first a base application is developed using standard Java, and then the crosscutting aspects are implemented as add-on aspects. The AspectJ language is based on standard Java, with additional keywords for pointcuts, advice and aspects.

2.2 Hyper/J

Hyper/J [5] has been developed at the IBM research labs. It defines the notion of *hyperspaces* [6], in which concerns can be identified and encapsulated, related and finally integrated. Each application concern defines an orthogonal dimension in the hyperspace. As a result, the Hyper/J approach is *symmetric*, as opposed to the "base application and aspect add-ons" approach taken by AspectJ.

In Hyper/J, methods and fields are considered primitive units, while interfaces, classes and packages are considered compound units. Each dimension of concern is made up of a set of units, either primitive or compound. A dimension of concern that *is self-contained* — in other words a dimension in which all member methods and variables that are referred to by any other member method are at least defined abstractly — can be split up into a *hyperslice*. In order to create a running application (called a *hypermodule*), the developer must define how the hyperslices are assembled in a configuration file.

Concern implementations in Hyper/J are written in standard Java. The implementation of our experiment has been compiled with Hyper/J version 1.2. The development of Hyper/J is no longer continued, but a more advanced Concern Manipulation Environment [7] is under development.

2.3 AspectWerkz

AspectWerkz [8], originally created by Jonas Boner and Alexandre Vasseur, is a dynamic and lightweight AOP framework for Java. The language is heavily inspired by AspectJ, in the sense that it uses the same concepts, namely pointcuts, advice, and introductions. However, AspectWerkz does not extend the Java language; aspects are just plain Java classes. Pointcuts are defined inside Java comments using an AspectJ-like syntax, or inside an accompanying XML file. Finally, an XML configuration file tells the AspectWerkz runtime which classes should be considered aspects. The version of AspectWerkz used for this experiment is 0.10, the most recent version available is 1.0RC3.

2.4 Technology

It is interesting to compare how the different environments implement weaving. The technology used has an impact on the performance, and on the expressiveness of concerns in general.

AspectJ, for example, extends Java with new language constructs, and thus compile-time weaving is most suitable when producing the final application. In fact AspectJ weaves aspects at compile time in 2 phases: front-end weaving and back-end weaving. The front-end weaving compiles both aspects and regular Java classes into pure Java bytecode annotated with additional attributes representing non-Java parts, such as advice, pointcuts etc. The back-end weaving transforms with the help of these additional attributes the pre-compiled bytecode generated by the front-end into standard Java bytecode. This transformation involves checking for each potential join point if there is a matching advice, and if there is, a call to that advice code is inserted. Since the final weaving is done at the bytecode level, join points in classes for which source code is not available can be handled, too.

In Hyper/J, weaving is just a process of concern composition, which most of the time boils down to merging several partial class views implementing a single concern into a single class implementing several concerns. The Hyper/J weaver works on Java class files, so it implements bytecode weaving just like AspectJ.

From a technological point of view, AspectWerkz is very interesting. It performs dynamic weaving, meaning that it is possible to dynamically add, remove and restructure advice as well as swapping the implementation of the introductions at run-time. AspectWerkz is based on run-time bytecode modification. One of the most interesting features of Aspectwerkz is its ability to run in two different modes: online and offline. In online mode, AspectWerkz is hooked into the low-level classloading mechanism part of the JVM, allowing it to intercept all classloading calls and transform the loaded bytecode on the fly. In order to make that happen, a wrapper script can be used as a replacement for the bin/java command to automatically perform the necessary customization depending on the Java version and JVM capabilities. This mode holds many benefits to developers, as it weaves classes at classload time, which means that class files can be deployed as usual. However, it does require an additional configuration of the application server, which may not be possible in some situations. In offline mode, two phases are required to generate the final woven classes. The first phase is the standard compilation using the *javac* tool. The second phase is to run the AspectWerkz compiler in offline mode and point it to the newly created class files. The compiler will then modify the bytecode of the classes to include advice calls at the pointcuts specified in the XML file. The benefit of using the offline mode is that the classes will run in any JVM.

2.5 Language Features

This subsection compares how the different approaches expose aspect-oriented concepts to the developer at the programming language level.

Concern Modules

Each of the approaches uses different modules for encapsulating a concern. AspectJ extends Java by adding a new construct, the aspect. Very often, a concern can be implemented in its entirety inside a single aspect. The timing and billing aspects, for instance, have been nicely encapsulated inside a single source file containing one aspect. In more complicated situations, for instance when trying to write reusable aspects, more than one aspect might be necessary to implement a concern. The persistence aspect, for instance, is composed of two aspects and an interface.

Conceptually, in Hyper/J, a concern is implemented in a *hyperslice*. The actual code can be spread over any number of classes. It is good practice to group all classes implementing a concern in a package. What files make up a concern is specified in the concern mapping section of the configuration file (see Figure 4 for an example).

In AspectWerkz, a concern is implemented in a standard Java class. As soon as introductions are used, however, a separate

mixin interface must be declared, which is then subsequently implemented in a static inner class.

Join Points

The AspectJ and AspectWerkz join point models are close to identical. Both approaches can perform weaving at method execution and call, constructor execution and call, field access, field modification, and exception handler execution. They both also support the *cflow* construct, which allows the developer to use call graph information when selecting join points. Hyper/J only defines method and constructor calls as join points.

Specifying Static Structure

To introduce new fields or methods into a class, AspectJ provides so-called introductions or inter-type declarations. Any standard field or method declaration in an aspect that is prefixed by a class name effectively injects that field or method into the class. AspectWerkzs introduction mechanism is more complicated. A static inner class (also called mixin class) has to be defined inside the aspect, together with an interface declaration. This inner class is then mixed into the target class. In Hyper/J, specifying static structure is done by simply defining those parts of a class that are important for a certain concern. During the weaving process, all parts of a class that conceptually constitute the same class are *merged* into a single class.

Specifying Behavior

The code that describes the behavior of a concern in all three approaches is just standard Java code. It is encapsulated in *methods* in Hyper/J, *annotated methods* in AspectWerkz, and in *advice* in AspectJ. In AspectJ and AspectWerkz, this code can be specified to execute *before*, *after* or *around* a join point. In Hyper/J, code of several methods can be *merged*, i.e. appended one after the other, or arranged in a certain order using the *bracket* construct in the configuration file (see Figure 20 as an example).

In certain situations, the code implementing behavior of a concern must get access to information available during run-time when the actual join point is reached. AspectJ defines a very elaborate constructs, such as `target()`, `args()`, `this()`, that allow a join point to pass parameters to the advice code (see Figure 3 as an example). AspectWerkz 0.10 only passes one parameter, a join point object, to the code[1]. This join point object, however, provides methods to query run-time information, such as the target object of a call or the value of an argument, when needed (see Figure 5 as an example). In Hyper/J the run-time information that can be passed to concern code is fairly limited. Only passing the called method name, or class name, or value of an argument is supported.

3. THE TELECOM APPLICATION

The application used for this experiment is a simple telecommunication simulation program. It emulates customers making local and long distance calls.

The original telecom application is part of the example applications that come with the AspectJ distribution. The base application that provides standard and conference calls consists of five simple classes as depicted in the UML class diagram shown in Figure 1.

[1] The current version of AspectWerkz provides more elaborate advice parameters.

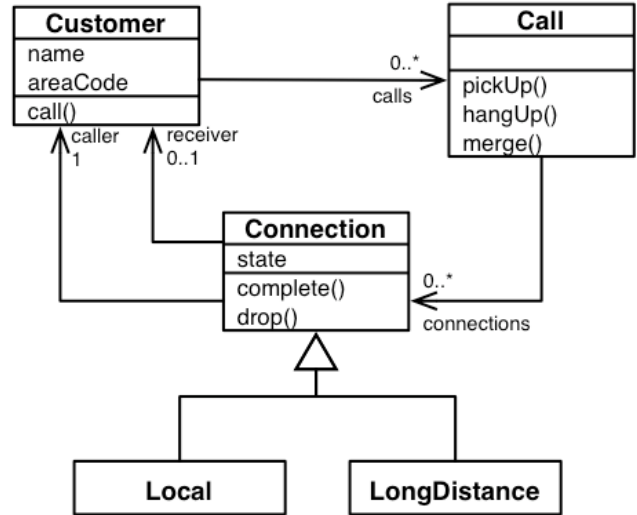

Figure 1: The Static Structure of the Telecom Application

Figure 2 shows a sequence diagram that depicts how two customers, *Bob* and *Alice*, set up a standard phone call. The caller, Bob, first instantiates a new `Call` object `c1`, which in turn creates a new `Connection` object `c`. The class of the connection, local or long distance, depends on the area codes of the caller and the receiver. In order to accept the call, *Alice* has to invoke the `pickup()` method, which in turn calls the `complete()` method of the corresponding connection. At the end of the call, either one of the customers can invoke `hangup()`, which terminates the connection by calling `drop()`.

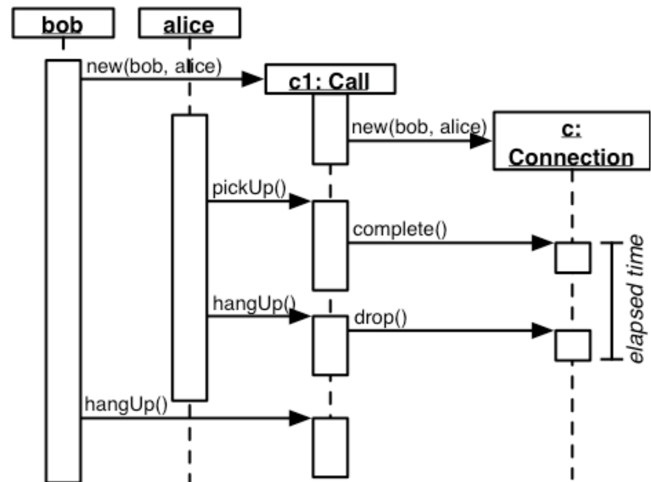

Figure 2: A Call between two Customers

Conference calls can be created by merging several calls together using the `merge()` method provided by the `Call` class.

4. TIMING ASPECT

4.1 Description

In a typical telecom application it is important to record for each call the elapsed time of the communication as well as the total connection time of each customer. A priori, timing is a separate concern that has nothing to do with how to establish a successful communication. Therefore, any aspect-oriented programming

environment should make it possible to modularize the *timing concern* and keep its implementation separate from the code that handles the details on how to establish a communication.

The basic timing functionality is offered by a class `Timer` that provides the self-explanatory methods `start()`, `stop()` and `getTime()`. In order to be correct even in the presence of conference calls, timing must be performed for each connection. The time that is to be measured is the elapsed time between the moment when the connection is established (the method `complete()` is called) and the moment the connection terminates (the method `drop()` returns (illustrated in Figure 2)).

In order to implement the timing concern, we essentially have to add four features to the base application:

1) A data structure that stores the total connection time for each customer,
2) A timer for each connection,
3) Behavior that starts the timer when a connection is established,
4) Behavior that stops the timer when the connection is dropped.

4.2 Timing in AspectJ

In AspectJ, all the modifications to the base classes of the telecom application that are related to the timing concern, i.e. the four features mentioned at the end of section 4.1, can be grouped into one aspect in a single source file. As a result, the obtained code is short and to the point, as illustrated in Figure 3.

```
public aspect Timing {

1 public long Customer.totalConnectTime = 0;
1 public long getTotalConnectTime(Customer cust)
1   { return cust.totalConnectTime; }

2 private Timer Connection.timer = new Timer();
2 public Timer getTimer(Connection conn)
2   { return conn.timer; }

3 after (Connection c):
3   target(c) && call(void Connection.complete())
3     { getTimer(c).start(); }

4 after (Connection c): target(c) &&
4       call(void Connection.drop())
4   { getTimer(c).stop();
4     c.getCaller().totalConnectTime +=
4       getTimer(c).getTime();
4     c.getReceiver().totalConnectTime +=
4       getTimer(c).getTime(); }
}
```

Figure 3: AspectJ Code for the Timing Concern

The lines tagged with 1 show how a so-called *inter-type declaration* can be used to extend the base `Customer` class, adding a new public field `totalConnectTime` and a corresponding getter method. In a similar way, the lines tagged with 2 add a new timer field to the base `Connection` class and initialize it with a new `Timer` object. Again, a getter method for the timer field is defined.

The behavior of starting the timer is added using the lines tagged with 3. The pointcut states that after any call to the `complete()` method of the `Connection` class, the timer associated with the called connection is started. In AspectJ, the target of a method call can be obtained from the run-time environment using the `target()` designator. Finally, in a similar way, after returning from a call to the `drop()` method, the timer is stopped and the total connection time of the corresponding customers is updated (see lines tagged with 4).

4.3 Timing in HyperJ

Whereas the AspectJ code can be seen as a compact add-on to the base application, the Hyper/J code implementing the timing concern is more or less stand-alone. The code is located in two files, the `Customer` file and the `Connection` file, grouped inside the package `Timing`. In addition, a configuration file has to be written. The complete code is shown in Figure 4.

```
  package Timing;
1 public class Customer {
1   private long totalConnectTime = 0;
1   public long getTotalConnectTime()
1     { return totalConnectTime; }
1   public void addTotalConnectTime(long t)
1     { totalConnectTime += t; }
}

  package Timing;
2 public abstract class Connection{
2   private Timer timer = new Timer();
3   void complete()
3     { timer.start(); }
4   void drop() {
4     timer.stop();
4     getCaller().addConnectTime(timer.getTime());
4   getReceiver().addConnectTime(timer.getTime());
    }
5   abstract public Customer getCaller();
5   abstract public Customer getReceiver();
}

-hyperspace
  hyperspace TimingHyperspace
    composable class telecom.*;
    composable class timing.*;
-concerns
  package telecom : Feature.Telecom
  package timing : Feature.Timing
-hypermodules
  hypermodule TimingModule
    hyperslices:
      Feature.Telecom,
      Feature.Timing;
    relationships:
  *   mergeByName;
  end hypermodule;
```

Figure 4: Hyper/J Code implementing the Timing Concern

The `Customer` class, tagged with 1, implements the field that stores the `totalConnectionTime` and a method for updating that field. In a separate file, the `Connection` class is specified to instantiate a timer at creation time (2), start the timer in the `complete()` method, and stop the timer in the `drop()` method.

Hyper/J uses a separate configuration file to define the actual source files that consistute the different concerns. An example of such a file is shown at the end of Figure 4. The code first defines two hyperslices, stating that the telecom concern is implemented by all units found in the `telecom` package, and that the timing concern is implemented by all units in the `timing` package.

Finally, the hypermodule part of the configuration file instructs Hyper/J to take all classes of the two slices and *merge them by name* (see * in Figure 4), meaning that classes with the same name will be merged together, and such are methods with the same name. That way, the code of the `complete()` and `drop()` methods of the timing hyperslice is appended to the code of the corresponding methods of the telecom hyperslice.

It is interesting to note here that in order to update the total connection time of a customer, the `drop()` method has to obtain the *caller* and *receiver* customer objects of the connection. Although these methods are used from within `drop()`, they are not implemented in the timing hyperslice. They are just declared there (see lines tagged with 5) for what Hyper/J calls *declarative completeness*. Once merged, their functionality is implemented by the method bodies of the base telecom `Connection` class.

4.4 Timing in AspectWerkz

The code implementing the timing concern in *AspectWerkz* is depicted in Figure 5. Note that the way of grouping all features together is by just simply enclosing all the code inside a class called `Timing`.

```
public class Timing {

1 /**@Introduce class(telecom.Customer)
1               deploymentModel=perInstance  */
1 public static class CustomerIntroduction
1   implements CustomerConnectTimeMixin {
1   private long totalConnectTime = 0;
1   public long getTotalConnectTime()
1     {  return totalConnectTime;  }
1   public void setTotalConnectTime(long time)
1     { totalConnectTime = time;  }
1 }

2 /**@Introduce class(telecom.Connection+)
2               deploymentModel=perInstance  */
2 public static class ConnectionIntroduction
2   implements ConnectionTimerFieldMixin {
2   private Timer timer = new Timer();
2   public Timer getTimer() {
2     {  return timer;  }
2 }
2 public static Timer getTimer(Connection conn) {
2    return ((ConnectionTimerFieldMixin)conn).
2            getTimer();
2 }

3 /**@After
3   call(void telecom.Connection+.complete()) */
3 public void afterComplete(JoinPoint joinPoint) {
3   Connection c = (Connection)joinPoint.
3                   getTargetInstance();
3   getTimer(c).start();
3 }

4 /**@After call(void telecom.Connection+.drop())
4  */
4 public void afterDrop(JoinPoint joinPoint) {
4   Connection c =(Connection)joinPoint.
4                 getTargetInstance();
4   getTimer(c).stop();
4   CustomerConnectTimeMixin customerMixin =
4         (CustomerConnectTimeMixin)c.getCaller();
4   customerMixin.setTotalConnectTime(
4         customerMixin.getTotalConnectTime() +
4         getTimer(c).getTime());
4   customerMixin =(CustomerConnectTimeMixin)c.
4         getReceiver();
4   customerMixin.setTotalConnectTime(
4     customerMixin.getTotalConnectTime() +
4     getTimer(c).getTime());
4 }
}
```

Figure 5: AspectWerkz Code for the Timing Concern

To distinguish the `Timing` class from other regular Java classes, a separate XML configuration file has to be written that tags the `Timing` class as being an aspect. An example XML file for the Timing aspect is shown in Figure 6.

```
<aspectwerkz>
   <system id="Telecom">
      <aspect class="telecom.TimingAspect"/>
</aspectWerkz>
```

Figure 6: XML Configuration File for AspectWerkz

AspectWerkz uses inner classes to introduce new fields into regular classes. This requires some additional work from the programmer: an inner class has to be written for each new field, and *getter* and *setter* methods have to be provided that read and modify the field. In addition, an *interface* exposing the getter and setter methods has to be defined.

As an example, the following interface is defined in order to add the `totalConnectionTime` field into the `Customer` class:

```
public interface CustomerConnectTimeMixin {
  public long getTotalConnectTime();
  public void setTotalConnectTime(long time);
}
```

Figure 7: CustomerConnectTimeMixin for AspectWerkz

Given this interface, the code tagged with 1 in Figure 5 introduces the `totalConnectTime` field and its access methods into the `Customer` class by declaring a new inner class that implements the `CustomerConnectTimeMixin` interface. The instructions that tell the aspect weaver *where* to introduce the class are given as Java comments that precede the method. For example, the first two lines tagged with 1 in Figure 5 state that the static class `CustomerIntroduction` implementing the `Customer-ConnectTimeMixin` interface should be woven into every instance of the `Customer` class.

Introducing additional behavior is done by declaring regular methods with a single `JoinPoint` parameter. AspectWerkz uses an AspectJ-like syntax to define where the code should be woven into the base application. For example, the code tagged with 3 in Figure 5 declares with the comment `@After call void telecom.Connection+.complete()` that the code of the `afterComplete()` method should be inserted after any call to the `complete()` method of a connection object or subclass.

In a similar way, the return of any call to the `drop()` method is intercepted in the lines tagged with 4. What is interesting to note here is that the fields introduced by an aspect can not be used directly, not even from within the same aspect. The object has to be cast to the interface which declares the getter and setter methods. In our example, when a call is completed, the customer object that represents the caller has to be cast to the `CustomerConnectTimeMixin` interface in order to access the `totalConnectTime` field.

5. BILLING ASPECT

The billing concern is responsible for billing customers for the calls they place. There are two types of calls, local and long distance. Both have different costs associated with them and must therefore be handled differently.

The billing concern is interesting from a conceptual point of view because in order to calculate the correct price for a call, the implementation must access information from the base application, such as the kind of call, but also information from the timing aspect, namely the connection time of the call.

In order to keep track of the billing information, our aspect implementation must add the following features to the base application:

1) Information about the rates for local and long distance calls,
2) Behavior that intercepts the end of a call and calculates the total cost based on the information provided by the *timing* concern,
3) A data structure that keeps track of the total bill amount for each customer.

5.1 Billing in AspectJ

In an object-oriented way of doing things, different rates for local and long distance can be implemented elegantly using inheritance. An abstract method `getRate()` is added to the `Call` class, and then the two subclasses `Local` and `LongDistance` implement it. This is exactly what the following lines of AspectJ code do:

```
public abstract long Connection.getRate();
public long LongDistance.getRate()
  { return LONG_DISTANCE_RATE; }
public long Local.getRate()
  { return LOCAL_RATE; }
```

Figure 8: Introducing Call Rates in AspectJ

Calculating the cost of a call must happen after a connection is dropped. Again we can use an *after* advice such as the one shown below:

```
after(Connection conn): Timing.endTiming(conn) {
  long time = Timing.aspectOf().
    getTimer(conn).getTime();
  conn.caller.addCharge(conn.getRate() * time);
}
declare precedence: Billing, Timing;
```

Figure 9: Injecting Billing Behavior in AspectJ

Figure 9 illustrates how dependencies between aspects can be handled. First of all, the advice is attached to the `endTiming` pointcut defined in the `Timing` aspect, which guarantees that the billing happens together with timing. In order to make sure that the billing happens after the timing, a precedence declaration has to be made (last line of Figure 9).

In order to get the time of the call, the code obtains an instance of the timing aspect, and then with its help gets the timer object associated with the call object. The time from the timer is then multiplied with the call rate, and added to the total bill of the calling customer.

Adding a total charge field and the `addCharge()` method to a customer is done using an inter-type declaration, in the same way as it was done for the total connection time in Figure 3 of section 4.2, and therefore the code is omitted.

5.2 Billing in Hyper/J

The `Billing` package contains the classes that implement the billing concern in Hyper/J, namely `Customer`, `Connection`, `Local` and `LongDistance`. Defining a total bill field for a customer is similar to what has been done in section 4.3 when defining the total connection time field, and therefore this code is omitted.

The implementation of the `Connection` class, however, is really interesting and shown in Figure 10.

```
package billing;
3 import timing.Timer;

public abstract class Connection {

4 private Timer timer;
5 public Customer caller;

1 abstract long getRate();

2 void drop() {
2   caller.addCharge
2     (timer.getTime() * getRate());
  }
}
```

Figure 10: Implementing Billing in Hyper/J

Information about the rates for local and long distance calls is encoded in a similar way to what has been done in AspectJ. An abstract method `getRate()` is defined in this billing-oriented view of the `Connection` class (see line tagged with 1), and the `Local` and `LongDistance` subclasses provide the concrete realizations (see Figure 11).

```
public abstract class Local extends Connection {
  public static final long LOCAL_RATE = 3;
  long getRate() {
    return LOCAL_RATE;
  }
}
```

Figure 11: Encoding Call Rates in Hyper/J

The functionality of getting the connection time, calculating the price and updating the total bill of the customer is implemented in the `drop()` method (lines tagged with 2 in Figure 10). Note that we are still using the *mergeByName* weaving technique, and hence this code will be combined with the code from the *timing* hyperslice and the *telecom* hyperslice. Since the code uses the `Timer` class, we have to include it as shown in the line tagged with 3. We must also assume that there is a timer object declared inside each connection object, so we mention the declaration in line 4. This declaration is merged by the weaver with the one from the timing hyperslice, and will therefore at run-time contain a reference to the instantiated timer. Finally we need a reference to the customer object representing the customer who has made the call, for only he should be billed for the communication. Again, the `caller` field provides that information, and it is set up in the telecom concern, so it in enough to declare it (see 5 in Figure 10).

The total charge field and the `addCharge()` method for the `Customer` class are implemented in a separate stand-alone class, which is merged with the timing customer and the telecom customer at weave-time. This customer code has been omitted, since it is very similar to what is done in line 1 of Figure 4.

5.3 Billing in AspectWerkz

Call rate information in the AspectWerkz implementation of the billing concern is encoded just as in the AspectJ one. An interface defining the method `getRate()` is injected into the `Connection` class using the *@implements* construct, and the implementations of that method into the subclasses `Local` and `LongDistance` as shown in Figure 12.

```
/** @Implements class(telecom.Connection)
                deploymentModel=perInstance  */
public long getRate;

/** @Introduce class(telecom.Local)
               deploymentModel=perInstance   */
public static class LocalIntroduction
   implements ConnectionMixin {
   public long getRate() {
      return LOCAL_RATE;
   }
}

/** @Introduce class(telecom.LongDistance)
               deploymentModel=perInstance   */
public static class LongDistanceIntroduction
   implements ConnectionMixin  {
   public long getRate() {
      return LONG_DISTANCE_RATE;
   }
}
```

Figure 12: Introducing Call Rates in AspectWerkz

The total charge field is introduced into the Customer class by defining a `CustomerChargeMixin`. The corresponding source code has been omitted, since it is similar to what is done in line 1 of Figure 5.

The code that calculates the cost of a call is implemented in an after advice that is executed after a call to the `drop()` method. The source code is shown in Figure 13. From the target connection object and with help of the timing aspect, a reference to the associated timer object is found. The call rate is obtained using the `getRate()` method, and finally, the calculated cost is saved by means of the `CustomerChargeMixin`.

```
/** @After call(
           void telecom.Connection+.drop())   */
public void afterDrop(JoinPoint joinPoint) {
   Connection conn =
        (Connection)joinPoint.getTargetInstance();
   long time =
        TimingAspect.getTimer(conn).getTime();
   long rate = ((ConnectionMixin)conn).getRate();
   long cost = rate * time;
   CustomerChargeMixin customerMixin =
        (CustomerChargeMixin) conn.caller; 2
   customerMixin.setTotalCharge(
        customerMixin.getTotalCharge() + cost);
}
```

Figure 13: Injecting Billing Behavior in AspectWerkz

Obviously, timing has to occur before billing aspect, since the billing concern needs the duration of the call to compute the cost.

In AspectWerkz, the aspect execution order at a given join point depends on the order of aspect declaration in the configuration file. Aspects with low priority must be positioned above aspects with high priority. The configuration file of the telecom application including timing and billing is shown in Figure 13.

```
<aspectwerkz>
  <system id="Telecom">
     <aspect class="telecom.BillingAspect"/>
     <aspect class="telecom.TimingAspect"/>
                 … … …
</aspectwerkz>
```

Figure 14: Declaring Aspect Precedence in AspectWerkz

6. PERSISTENCE ASPECT

In [9] persistence is defined as follows:

Persistence is the property of an object through which its existence transcends time (i.e. the object continues to exist after its creator ceases to exist) and / or space (i.e. the object's location moves from the address space in which it was created).

In practice, persistence is used whenever data values from one program execution are saved for a later execution. The most sophisticated and desired form of persistence is *orthogonal persistence* [10]. It is the provision of persistence for all data, irrespective of their type. In a programming language providing orthogonal persistence, persistent data is created and used in the same way as non-persistent data. Loading and saving of values does not alter their semantics, and the process is transparent to the application program. Whether or not data should be made persistent is often determined using a technique called *persistence by reachability*. The persistence support designates an object as a persistent root and provides applications with a built-in function for locating it. Any object that is "reachable" from the persistent root, for instance by following pointers, is automatically made persistent.

But transparently saving and loading data from external storage devices can slow down an application significantly. In certain situations it makes sense to explicitly tag important data that is to be made persistent, and explicitly state when the data is to be written to the storage device. That way, the persistence support does not by accident persist transient data, or data that does not have to survive program termination.

This is the approach followed in our telecom application. The important data from an application perspective, i.e. the data that must survive program termination, is the total connection time and total bill of each customer. For each customer, the value of these two fields has to be saved to disk, and loaded from disk when the application is (re-)started. For simplicity, our code uses a file to store the data. A more sophisticated approach is to store the data on stable storage [11], for instance, in a database. The interested reader is directed to [12], which shows how AspectJ can be used to interface with a SQL database.

The persistence concern is particularly interesting due to the fact that persistence is a fairly common concern for many applications. An aspect-oriented programming technique that allows a developer to implement a certain concern in a general way, and then apply it in different contexts opens a lot of reuse possibilities. In order to evaluate the reuse potential of the different approaches, the code implementing persistence in the following sections has been designed to be as general and reusable as possible. For

[2] In our implementation, a type casting exception is thrown whenever customer object is cast to the `CustomerChargeMixin` interface. After several attempts, we concluded that this must be a bug in the AspectWerkz 0.10 release.

simplicity, all following implementations make the assumption that the class that is to be made persistent defines *setter methods* for each and every field, and that fields are only modified using the setter methods.

6.1 Persistence in AspectJ

General Implementation

A simple way of tagging objects that are to be made persistent is to provide an abstract `PersistentObject` class. In order to create an application-defined persistent object, the programmer has to extend the `PersistentObject` class, adding application-specific fields. This approach has proven to be very effective. It has been used extensively in frameworks providing persistence [13] [14] or even in aspect-oriented approaches such as [12].

```
abstract public class PersistentObject {
   abstract public String _getFileName();
}
```

Figure 15: PersistentObject class in AspectJ

Figure 15 shows the abstract class that all application-specific persistent classes should inherit from. It defines an abstract method `_getFileName()` that returns the filename that should be used for storing the data.

The actual implementation of persistence in AspectJ is done in an aspect as shown in Figure 16. The aspect defines two pointcuts: one that intercepts calls to the constructor of a persistent object (see line 1), and calls to setter methods (see line 2). After the constructor has executed, the after advice tagged with 3 runs. For space reasons, the detailed code can not be shown in this paper, but the main idea is that all fields are read from the corresponding file, and the values are then copied into the newly created object using the Java reflection API. This standard technique works fine for fields of the base `Customer` class. It is slightly more complicated for fields injected into the base class by inter-type declarations, such as `totalConnectTime` field which has been introduced by the `Timing` aspect. AspectJ changes the name of fields when introducing them into a base class following a specific pattern, mainly to avoid name conflicts. Our code has to handle these fields accordingly.

```
privileged aspect Persistence {

1 pointcut instantiation():
    call(PersistentObject+.new(..));

2 pointcut setters(PersistentObject o): target(o)
    && call (void PersistentObject+.set*(..));

3 after() returning (PersistentObject o):
3   instantiation() {
3   // code that loads all fields from the file
3   // using the Java reflection API
  }

4 after(PersistentObject o): setters(o) {
4   // code that writes the field that was
4   // modified to the file
  }
}
```

Figure 16: Persistence in AspectJ

Customization

Finally, to make the `Customer` class persist, an aspect has to be written that adds the `PersistentObject` class as a parent of the `Customer` class, and implements the `_getFileName()` method. This is shown in Figure 17. The desired inheritance relationship is established using the *declare parents* construct of AspectJ.

```
public aspect CustomerPersistence {

  declare parents:
    Customer extends PersistentObject;

  public String Customer._getFileName() {
    return this + ".dat";
  }
}
```

Figure 17: Making Customer Persist in AspectJ

6.2 Persistence in Hyper/J

General Implementation

Persistence in Hyper/J is implemented in a seperate package `Persistence`. Figure 18 shows the persistence concern implemented in a reusable way in a single class.

```
public abstract class PersistentObject {

1 abstract protected String _getFileName();

2 public void init() throws … {
2   // code that loads all fields from the file
2   // using the Java reflection API
  }

3 public void afterSetLong(long value)
3   throws IOException {
3   // code that writes the field that is to be
3   // modified to the file
  }
  …
}
```

Figure 18: Persistence in Hyper/J

The `init()` method (see lines tagged with 2) is the one that takes care of loading all field values from the save file after the constructor of the persistent class executes. The code is similar to the one used in the AspectJ, i.e. it uses reflection to initialize all the fields. `AfterSetLong` (see lines tagged with 3) is capable of writing a `long` value to the file. It will execute whenever a setter method for a field of type `long` returns. The `PersistentObject` class contains similar methods for every kind of basic Java type.

Customization

The persistence class is abstract since it defines an abstract `_getFileName()` method (see line 1). Any class that wishes to make its state persistent must provide an implementation for it. Figure 19 shows how this is done for the `Customer` class in the package `Persistence.Impl`.

```
public class Customer {
  protected String _getFileName() {
    return this + ".dat";
  }
}
```

Figure 19: Persistence for Customers in Hyper/J

Finally, to put the pieces of the puzzle together, the developer must write a hypermodule configuration file such as the one shown in Figure 20.

```
hypermodule PersistenceModule
  hyperslices:
    Feature.Telecom,
    Feature.Timing,
    Feature.Billing,
    Feature.Persistence,
    Feature.PersistenceImpl;
  relationships:
1   mergeByName;
2   equate class
      Feature.Telecom.Customer,
      Feature.Timing.Customer,
      Feature.Billing.Customer,
      Feature.Persistence.PersistentObject,
      Feature.Persistence.Impl.Customer;
3   bracket "Customer"."<init>" with after
      Feature.Persistence.PersistentObject.init;
4   bracket "Customer"."set*" with
      after Feature.Persistence.
        PersistentObject.afterSetLong;
  end hypermodule;
```

Figure 20: Persistence Configuration for Hyper/J

The general weaving instructions for the persistence concern are again *mergeByName* (see line 1), just as for the *Timing* aspect. Persistence, however, has been implemented as a stand-alone, reusable concern (as opposed to implementing persistence for customers directly). For achieving persistence for `Customer` objects, the developer must explicitly state that the customer class is also a persistent class, or, in Hyper/J terms that the four different `Customer` implementations and the `PersistentObject` class must be merged into one and the same class. This is done using the *equate class* construct as shown in line 2. Finally, the constructor and setter calls have to be intercepted. This is done using the *bracket* construct of Hyper/J. Line 3 states that the `init()` method of the persistence class should execute after any customer constructor call. Line 4 states that `afterSetLong` should be executed after any call to a setter of a customer field of type `long`. Similar backet statements have to be defined for each base Java type. Hyper/J will dispatch to the correct method based on the parameter of the setter method.

6.3 Persistence in AspectWerkz

AspectWerkz does not provide a primitive that allows a developer to change the inheritance hierarchy (such as the `declare parents` primitive in AspectJ). Therefore it is not possible to create a `PersistenceObject` class from which all persistent objects should inherit (see section 6.1). The alternative is to create an instance of the persistence aspect for every instance of `Customer`.

General Implementation

The general persistence implementation in AspectWerkz is depicted in Figure 21.

```
public class PersistenceAspect {
1 /* @Around persistentObjectInitialized */
1 public Object afterConstructor
1   (JoinPoint joinPoint) throws Throwable {
1   Object o = joinPoint.proceed();
1   …
  }
2 /* @Before persistentObjectUpdated */
2 public void beforeSetOperation
2   (JoinPoint joinPoint) throws IOException {
2   Object o = joinPoint.getTargetInstance();
2   MethodRtti mRtti =
2     (MethodRtti)joinPoint.getRtti();
2   Object value = mRtti.getParameterValues()[0];
2   …
  }

3 public String _getFileName(Object o) {
3   throw new NameNotDefinedException();
  }
}
```

Figure 21: Persistence in AspectWerkz

The lines tagged with 1 handle the interception of constructor calls. An `around` pointcut combined with `proceed()` is used in order to obtain a reference to the newly created object. Then the fields can be initialized with the values read from the file (code not shown due to space reasons). The code tagged with 2 intercepts updates, gets the target object of the update, gets a representation of the method, gets the value of the first parameter, and saves the new value to the file.

In order to get a reusable implementation of persistence, the `Persistence` class defines the named pointcuts `persistentObjectInitialized` and `persistentObjectUpdated`, but does not specify which join points they correspond to. Also, the filename of the file to be used, obtained from the `_getFileName()` method, is not known at this point.

Customization

In order to make the `Customer` class persist, the `Persistence` class must be extended, the actual join points specified, and the `_getFileName()` method overridden as shown in Figure 22.

```
public class CustomerPersistenceAspect
  extends PersistenceAspect {

1 /* @Expression call(telecom.Customer.new(..)) */
1 Pointcut persistentObjectInitialized;

  /* @Expression */
2    call(void telecom.Customer.set*(..) )*/
2 Pointcut persistentObjectUpdated;

3 public String _getFileName(Object o) {
3   return o + ".dat";
  }
}
```

Figure 22: Making Customer Persist in AspectWerkz

The lines tagged with 1 specify to intercept calls to constructors of `Customer`. The lines tagged with 2 specify to intercept calls to setter methods. The lines tagged with 3 define the filename of the file to be the string value of the customer object, i.e. its name, with the extension "`.dat`".

To produce the final application with persistent customers, the XML configuration file must be updated to include `PersistenceAspect` and `CustomerPersistenceAspect` in the set of aspects to be considered by the weaver. In order to get a separate instance of the aspect for each customer, the *perInstance* deployment-model must be specified as shown in the last two lines of Figure 23.

```
<aspectwerkz>
  <system id="Telecom">
     <aspect class="telecom.TimingAspect"/>
     <aspect class="telecom.BillingAspect"/>
     <aspect
class="telecom.CustomerPersistenceAspect"
deployment-model="perInstance"/>
     <aspect class="telecom.PersistenceAspect"
deployment-model="perInstance"/>
    </system>
</aspectwerkz>
```

Figure 23: Configuring Persistence

It is interesting to note here that it was impossible to define an abstract method `_getFileName()` in the `Persistence` class, because this would have made the class itself abstract. The way aspect-orientation is implemented in AspectWerkz, all aspects are instantiated during run-time. Another technically interesting point is the fact that the customer object must be passed explicitly to the `_getFileName()` method as a parameter. This comes from the fact that in AspectWerkz aspects do not get woven into the target class, but keep a separate identity. Hence, when executing advice code, `this` refers to the aspect instance, not to the corresponding object. This is different from both AspectJ and Hyper/J, where abstract methods can be used, and where inter-type declarations can augment classes / merging of classes is possible.

7. DISCUSSION

The detailed descriptions from the previous sections show that all three approaches were successful in modularizing the four application concerns. None of the resulting designs is clearly superior to the others. Each one has advantages and disadvantages. This section compares the approaches according to the conceptual and actual programming effort they require, and the reuse potential of concern implementations.

7.1 Ease of Use

After having implemented the same application in three different aspect-oriented programming environments, it makes sense to compare the overhead each approach imposes on the programmer. Clearly, AspectJ produces the most compact code. Simple concerns, such as the timing concern, can be implemented inside a single aspect. Hyper/J concerns tend to be implemented in many small classes and thus require more typing, but are grouped using Java packages. The resulting structure is easy to maintain, even when the number of concerns of an application increases. AspectWerkz requires to write lengthy interface definitions and mixin classes when introducing static structure into classes, which is a burden for the programmer, and leads to less readable code. However, an appropriate tool might be able to automate the generation of these artifacts.

Our experiments also confirmed the findings described in [8] and [18], namely that asymmetrical AOP approaches such as AspectJ and AspectWerkz are easy to adopt. The incremental design approach, starting with a base application and adding concerns one by one, seems more natural to programmers that are used to standard object-orientated development. Hyper/J initially requires a certain effort from developers. They have to “switch” to the symmetrical programming paradigm. However, once acquired, the symmetrical approach leads to a simpler, more readable code structure. Understanding the weaving process from a conceptual point of view is also easier, since Hyper/J uses *merging* for both composing static structure and for composing run-time behavior.

Ease of use is also greatly influenced by the actual *programming interface*. Especially aspect-oriented programming can benefit from a good integration with the development environment [16]. AspectJ, for instance, provides plug-ins for all major development environments. They enable the environment to visualize for a given aspect which join points in the base code it affects. AspectWerkz only provides a plug-in for NetBeans, and to our knowledge, Hyper/J does not offer tool support at all. This situation might change soon, however, with the development of the Concern Management Environment [7].

Finally, another important factor for ease of use is *safety*. An implementation of one concern might have an impact on other concern implementations. For instance, for concerns that apply to the same join point, a well thought-through precedence ordering has to be established. The lack of an explicit ordering statement among conflicting concerns can be misleading for an application developer and may lead faulty program behavior, especially if a default ordering is implicitly applied by the weaver.

7.2 Reusability

All three AOP approaches were successful in implementing the persistence concern in a general, reusable way. However, there are important structural differences.

AspectJ implements general persistence with an interface and an aspect that intercepts constructor and setter methods; the customization step requires another aspect that sets up the *implements* relationship between the class to persist and the interface. Hyper/J implements general persistence in a single abstract class; the customization step requires a concrete class and the definition of a concern mapping that designates the constructor and the setter methods. AspectWerkz implements general persistence in an aspect with abstract pointcuts; the customization step requires extending that aspect and defining the concrete pointcuts to designate the constructor and setter methods. A summary table of the structural differences is shown in Figure 24.

	General Persistence	Customization
AspectJ	Interface & Aspect with Behavior Mapping	Aspect with Static Mapping
Hyper/J	Abstract Class	Concrete Class & Merge Mapping
AspectWerkz	Aspect	Extension of General Aspect to Specify Behavior Mapping

Figure 24: Comparing the Persistence Implementation

The approach in AspectJ is safe, since it is not the customizing programmer’s responsibility to specify that constructor and setter method calls must be intercepted. The persistence aspect is already taking care of that, and therefore the programmer can not forget to do so. As a result, the customization aspect is short and simple to write. On the other hand, the Hyper/J and AspectWerkz approach are more flexible, since the customizing programmer can designate which methods are to be considered modifiers in

cases where the "`set*`" naming convention has not been followed[3].

In general, from a reuse perspective it is better to have the mapping be part of the customization step. Once a certain mapping has been committed to, it cannot be changed anymore. Highest flexibility is achieved when the mapping decisions are made as close as possible to weave-time, i.e. at the time the final application is composed. Only at weave-time the actual application configuration is known. In case of concern conflicts, it is important that the developer reusing a concern implementation be aware of all other concerns. Only by making sure that potential conflicts are resolved using precedence declarations, a correct application configuration can be established. For these reasons the existence of a separate configuration file is essential. AspectJ does not offer such a feature, whereas Hyper/J always relies on a separate configuration file. The desired effect can be achieved in AspectWerks if all pointcut, advice and aspect definitions are done in the XML configuration file.

8. RELATED WORK

[17] describes an exploratory study comparing AspectJ, Hyper/J and a lexical concern separation approach. In the experiment, two existing Java applications, gnu.regexp and jFTPd are analyzed and two specific concerns are identified. Then, the applications are refactored and re-designed using the three approaches to modularize the identified concerns. In the experiment the authors had to deal with separating concerns tangled within a method, and separating concerns tangled between classes. They conclude that different AOP approaches and mechanisms require different refactorings in order to be applicable, and finally lead to different application designs. They also provide a set of guidelines to help developers prepare their codebase for separating concerns.

[18] compares AOP approaches at a higher level, by discussing the software engineering properties of asymmetric and symmetric paradigms. The authors conclude that both approaches have advantages in simple extension scenarios (as the one described in this paper), but that the relative advantages of symmetrical approaches increase with the independence of the development effort in multi-extension and multi-component software. They also conclude that only symmetrical paradigms should be used as a basis of a reusable component model for software construction by composition.

There have also been several workshops at previous aspect-oriented conferences that look into aspect language and aspect modeling issues such as SPLAT (Software engineering Properties of Languages for Aspect Technologies) [19], FOAL (Foundations of Aspect-Oriented Languages) [20] and AOM (Aspect-Oriented Modeling) [21].

9. CONCLUSION AND FUTURE WORK

This paper describes an experiment, in which a single application has been implemented using three different aspect-oriented programming approaches. The experiment showed that the features offered by the aspect-oriented programming approaches under examination had an important impact on the design of the application. Based on this observation it is not far fetched to conclude that the presentation of aspect-oriented concepts at the language level greatly influences how developers nowadays design and structure aspect-oriented programs. Certain features might offer greater expressive power or more detailed control over the weaving process, but complicate the overall understanding of an application's structure and behavior. Some features make the programming activity itself less error-prone, and hence make the entire approach user-friendlier in general. Asymmetrical approaches might be easier to learn and adopt since they extend conventional programming, but on the other hand symmetrical approaches result in a simple and elegant concern structure even if the size of the application increases.

We believe that practical experiments such as the one presented in this paper are crucial for the future of aspect-oriented programming and aspect-orientation in general. They allows language designers and implementers of new aspect-oriented programming approaches to discover what language features have a direct impact on the usability of a language and on its software engineering properties.

In the near future we intend to implement the telecom application using other aspect-oriented programming approaches. A theoretical design has already been completed using the composition filters approach. However, the current state of the programming environment is not stable enough to validate the conceptual design by an implementation.

We intend to use the results of this experiment also in the field of aspect-oriented modeling. The aspect-oriented modeling community [15] strives to (graphically) model the "essence" of an aspect or concern. The three implementations shown in this paper implement the same application functionality in three different ways. We want to investigate if it is possible to model the intent of the individual concerns in a way that is independent of the actual aspect-oriented programming approach. If this is possible, plans are to define mappings that transform such a platform independent model into platform-, or in this case "aspect-oriented programming language"-specific models, applying the ideas of the Model Driven Architecture (MDA) [23].

[3] To be fair it must be said that it is possible to implement persistence in AspectJ in the latter way as well.

10. REFERENCES

[1] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr., " N degrees of separation: Multi-dimensional separation of concerns," in *Proceedings of the 1999 International Conference on Software Engineering*, Los Angeles, CA, USA, pp. 107 – 119, IEEE Computer Society Press, 1999.

[2] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher, "Discussing aspects of aop," *Communications of the ACM*, vol. 44, pp. 33–38, October 2001.

[3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *15th European Conference on Object–Oriented Programming*, June 18–22, 2001, Budapest, Hungary, pp. 327 – 357, 2001.

[4] E. Hilsdale and J. Hugunin, "Advice weaving in AspectJ," in *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development - AOSD'2004* (K. Lieberherr, ed.), pp. 26 – 35, ACM Press, March 2004.

[5] P. Tarr, H. Ossher, and S. Sutton, " Hyper/J: Multi-dimensional separation of concerns for Java," *in Proceedings of the 24th International Conference on Software*

Engineering, Orlando, Florida, pp. 734-737, IEEE Computer Society Press, May 2002.

[6] H. Ossher and P. Tarr, "Multi-dimensional separation of concerns and the hyperspace approach," in *Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, Kluwer Academic Publishers, April 2000.

[7] http://www.eclipse.org/cme/

[8] J. Bonér, "What are the key issues for commercial aop use - how does aspectwerkz address them?," in *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pp. 5 – 6, ACM Press, March 2004.

[9] G. Booch, *Object-Oriented Design with Applications.* Benjamin/Cummings Series in Ada and Software Engineering, Redwood City , CA , USA: The Benjamin/Cummings Publishing Company, Inc., 1991.

[10] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison, "An approach to persistent programming," *Computer Journal*, vol. 26, no. 4, pp. 360 – 365, 1983.

[11] B. W. Lampson and H. E. Sturgis, "Crash recovery in a distributed data storage system," Technical Report, XEROX Research, Palo Alto, June 1979.

[12] A. Rashid and R. Chitchyan, "Persistence as an aspect," in *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development - AOSD'2003*, (M. Aksit, ed.), pp. 120 – 129, ACM Press, March 2003.

[13] J. Kienzle and A. Romanovsky, "A framework based on design patterns for providing persistence in object-oriented programming languages*," IEE Proceedings - Software Engineering*, vol. 149, pp. 77 – 85, June 2002.

[14] J. Kienzle, *Open Multithreaded Transactions — A Transaction Model for Concurrent Object-Oriented Programming*. Kluwer Academic Publishers, 2003.

[15] Workshop Series on Aspect-Oriented Modeling. http://www.aspectmodeling.org/

[16] J. Kienzle and R. Guerraoui, "Aop - does it make sense? the case of concurrency and failures," in *16th European Conference on Object–Oriented Programming (ECOOP '2002) , Malaga, Spain*, Lecture Notes in Computer Science, Springer Verlag, 2002.

[17] G. C. Murphy, A. Lai, R. J. Walker, and M. P. Robillard, "Separating features in source code: An exploratory study," in *Proceedings of the International Conference on Software Engineering - ICSE 2001*, pp. 275 – 284, IEEE Computer Society Press, 2001.

[18] W. Harrison, H. Ossher, and P. Tarr, "Asymmetrically vs. symmetrically organized paradigmes for software composition," Tech. Rep. RC22685, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA, December 2002.

[19] SPLAT workshops. http://www.daimi.au.dk/~eernst/splat04/

[20] FOAL workshops: http://www.cs.iastate.edu/~leavens/FOAL/index-2004.shtml

[21] AOM workshops. http://www.cs.iit.edu/~oaldawud/AOM/

[22] M. Aksit, L. Bergmans, and S. Vural, "An object-oriented languagedatabase integration model: The composition-filters approach," in *6th European Conference on Object–Oriented Programming (ECOOP'92)* (O. L. Madsen, ed.), no. 615 in Lecture Notes in Computer Science, (Utrecht, The Netherlands), pp. 372 – 395, Springer Verlag, June 1992.

[23] S. J. Mellor, K. Scott, A. Uhl, and D. Weise, *MDA Distilled: Principles of Model-Driven Architecture*. Boston: Addison-Wesley, 2004.