Modularizing Communication Middleware Concerns Using Aspects

Cristiano Amaral Maffort, Marco Tulio de Oliveira Valente

PUC Minas, Institute of Informatics
Anel Rodoviário Km 23,5 - Rua Walter Ianni, 255
31980-110 - Belo Horizonte - MG - Brazil
Phone: +55 (31) 3439-5204
maffort@gmail.com, mtov@pucminas.br

Abstract

Software engineers often rely on communication middleware platforms to design and implement distributed systems. However, middleware functionality is usually invasive, pervasive and tangled with business-specific concerns. In this paper, we describe an aspect-oriented distributed programming system that encapsulates middleware services provided by Java RMI and Java IDL. The proposed system, called DAJ, handles the basic service provided by such object-oriented middleware platforms, i.e., synchronous remote calls using call by-serialization and call by-remote-reference semantics. The paper documents our experience in using DAJ to modularize middleware concerns from three legacy distributed systems.

Keywords: *aspects; separation of concerns; distribution; middleware.*

1. INTRODUCTION

During the last two decades software engineers have often relied on communication middleware platforms to design and implement distributed systems. Middleware platforms, such as CORBA [34], Java RMI [36], and Web Services [1], encapsulate several details inherent to distributed programming, including communication protocols, data marshalling and unmarshalling, heterogeneity, service lookup, synchronization, and failure handling. Although they shield developers from networking details, middleware systems also require developers to follow their specific code conventions and protocols. For example, programmers must deal with middleware specific classes, interfaces, and remote exceptions. Moreover, usually middleware code impacts several classes of distributed systems and forces such classes to handle more than one concern simultaneously. In other words, middleware code does not sit only between the network and the distributed application, but it is also spreaded and tangled in the latter [12, 29, 32]. The consequence is that distributed object-based systems do not present expected levels of separation of concerns, making such systems more difficult to implement, maintain and evolve.

This paper is a revised and extended version of a previous conference paper describing DAJ (*Distribution Aspects in Java*), a system designed to insulate middleware from business related code [23]. In order to accomplish its goal, DAJ relies on the synergistic combination of three technologies: aspects [16, 15], domain-specific languages [33, 3], and generative programming [8]. In the proposed system, aspects are used to encapsulate crosscutting middleware code, domain-specific languages are used to elevate the abstraction level when defining distributed software architectures, and generative programming is used to synthesize aspects and other components.

Particularly, DAJ includes an aspect-oriented framework that encapsulates crosscutting code required by communication middleware. Moreover, DAJ includes a generative tool that specializes abstract aspects from this framework in order to add distribution to a given application. The current version of the system generates aspects for two native JDK middleware platforms: Java RMI [36] and Java IDL [14]. Java RMI is a middleware often used in the implementation of Java-to-Java distributed applications. Java IDL adds CORBA support to the Java plat-

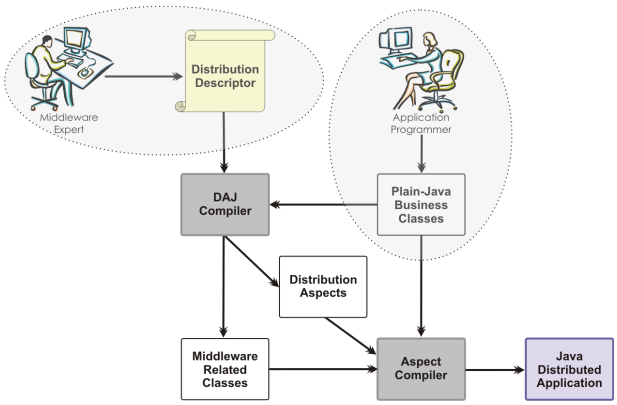

Figure 1. Developing distributed applications using DAJ

form, providing interoperability with distributed systems written in other programming languages.

In DAJ-based applications, business classes do not need to follow particular programming conventions. More specifically, they do not need to extend middleware pre-defined classes, implement remote interfaces or accessor and mutator methods; they also do not need to handle remote exceptions. Instead, a domain-specific language is used to declare the role that plain Java classes and objects play in a particular distributed system configuration. From specifications in such language, called distribution descriptors, DAJ generates classes and aspects that modularize the otherwise crosscutting middleware functionality. The aspects are generated in AspectJ [15, 19]. Figure 1 summarizes the steps proposed by the system to support the implementation of distributed object-based applications.

DAJ introduces at least two contributions to the conventional use of middleware platforms. First, and more important, developers can focus on the functional concerns of the system they are building since DAJ automatically generates aspects that modularize crosscutting code required by middleware platforms. Second, DAJ users do not need to master details and idiosyncrasies inherent to middleware APIs.

The remaining of this paper is organized as follows. Section 2 presents a motivating example, used in the rest of the paper to describe the DAJ system. Section 3 describes the programming interface that must be followed by DAJ users. Section 4 presents the software architecture and the implementation of the DAJ system, focusing on the abstract aspects defined by the system and the concrete aspects generated from the information defined in distribution descriptors. Section 5 describes our experience in using DAJ to modularize middleware concerns from three medium-sized distributed systems. This experimental study constitutes the main contribution of this work regarding our first paper about DAJ [23]. Section 6 presents a discussion about the lessons learned in DAJ development. Section 7 describes related work and Section 8 concludes the paper.

2. StockWatcher System

In order to describe the DAJ system, we will use in the remaining of this paper a distributed application that provides information about stock prices. The StockWatcher system consists of a remote object that stores the current price of a set of stocks. Clients access this object to update the price of a given stock and to subscribe for updates in the price of stocks. In the latter case, the server object uses a callback to notify the client.

Although simple, StockWatcher uses the core abstractions provided by object-oriented middleware systems. It relies on remote calls using call by-serialization and call by-remote-reference semantics when objects are used as parameters. A copy of an object is passed when call by-serialization semantics is specified (also known as object-by-value semantics in CORBA). In this case, there is no relation between the client-side object and the copy that the server uses during the execution of the remote method. On the other hand, when using call by-remote-reference semantics, the remote reference associated with the object is passed as argument of the remote call. During the execution of the remote method, this reference can be used to callback the object in the client address space.

StockWatcher admits several deployment configurations. For example, a deployment scenario may include a single RMI server and a number of Java clients. In another scenario, CORBA may be used in order to support interoperation with clients implemented in other programming languages. In a third scenario, we can have two servers, one implemented in CORBA and another implemented in Java RMI. In this case, each server handles a subset of the traded stocks.

We describe next the main interfaces and classes of the system.

Interfaces: The server that stores stock prices implements the following interface:

```
interface StockMarket {
  void update(StockInfo info);
  void subscribe(String stock,StockListener obj);
  void unsubscribe(String stock,StockListener obj);
  StockInfo getStock(String stock)
     throws StockNotFoundException;
}
```

Clients publish changes in the price of stocks by calling the `update` method. The class `StockInfo` is used to store stock prices in a given date and time:

```
class StockInfo {
  private String stock;
  private double value;
  private String date_time;
}
```

Clients register their interest in notifications of price changes by calling the method `subscribe`, passing the name of the stock they are interested in and a reference to a remote object used by the server to call back the client. The `unsubscribe` method is used to cancel the registration performed by `subscribe`. In both methods, the client object that the server calls back must implement the interface `StockListener`:

```
interface StockListener {
  void update(String stock, double value);
}
```

The `StockMarket` interface also provides the `getStock` method used to retrieve the current price of the stock passed as parameter. If the server does handle the requested stock, a remote exception of type `StockNotFoundException` is raised.

Implementation: In the implementation of the StockWatcher system, the class `StockMarketImpl` implements the interface `StockMarket` and the class `StockListenerImpl` implements the interface `StockListener`. Different from standard object-oriented implementations using Java RMI and Java IDL, the described interfaces and classes do not have any tangled code related to distribution.

3. PROGRAMMING INTERFACE

This section describes the DAJ programming interface. More specifically, we describe the conventions that developers must follow in order to define distribution descriptors and to program client and server modules.

3.1. DISTRIBUTION DESCRIPTORS

DAJ hides from business classes programming conventions demanded by the underlying middleware platform. The ultimate goal is to allow developers to focus on the functional requirements of the applications they are building, delegating to the framework the implementation of the aspects related to distribution concerns. However, developers must define the distributed configuration of their systems using an XML file called distribution descriptor.

Basically, distribution descriptors contain information about remote objects of a given distributed application. A remote object is one whose methods can be accessed from another address space, possibly from a different host. In DAJ, remote objects that are registered in a naming server are called server objects (or just servers). For such objects, DAJ users must define their names, classes and interfaces, the middleware used to access them, and the host and port number of the naming server where they must be registered[1]. Such information is specified using a `server` tag in the distribution descriptor. DAJ users must also define the types that are passed in remote calls using call by-serialization and call by-remote-reference. In the case of types passed by-remote-reference (node `remote`), developers must inform their interfaces and classes. In the case of types passed by-serialization (node `serializable`), developers must inform their classes.

Example: Figure 2 presents a possible distribution descriptor for the StockWatcher system. This descriptor assumes an architecture including two servers: the first uses Java IDL for communication and is registered under the name `StockMarketA` (lines 1 to 6); the second server uses Java RMI and is registered under the name `StockMarketB` (lines 7 to 12). Moreover, the descriptor prescribes that in remote calls objects of type `StockListener` are passed by-remote-reference (lines 13 to 16) and objects of type `StockInfo` are passed by-serialization (lines 17 to 19).

3.2. CLIENTS

In distributed object-based systems, clients locate remote objects and then perform remote calls using the same syntax of local calls. Clients must use the `getRef` method from the DAJ API to obtain references to server objects configured in a given distribution descriptor. Using such references, clients perform remote calls in a transparent way, i.e., without having any knowledge about the underlying middleware platform.

[1] A naming service allows clients to locate objects based on their names. The current implementation of DAJ is compatible with the standard naming servers provided by Java RMI and Java IDL, respectively called `rmiregistry` and `tnameserv`.

```
 1:  <server id="StockMarketA">
 2:    <interface>stockwatcher.StockMarket</interface>
 3:    <class>stockwatcher.StockMarketImpl</class>
 4:    <protocol>javaidl</protocol>
 5:    <nameserver>skank.pucminas.br</nameserver>
 6:  </server>
 7:  <server id="StockMarketB">
 8:    <interface>stockwatcher.StockMarket</interface>
 9:    <class>stockwatcher.StockMarketImpl</class>
10:    <protocol>javarmi</protocol>
11:    <nameserver>patofu.pucminas.br:1530</nameserver>
12:  </server>
13:  <remote>
14:    <interface>stockwatcher.StockListener</interface>
15:    <class>stockwatcher.StockListenerImpl</class>
16:  </remote>
17:  <serializable>
18:    <class>stockwatcher.StockInfo</class>
19:  </serializable>
```

Figure 2. A possible distribution descriptor for the motivating example

Example: We show next a fragment of a client module of the StockWatcher system:

```
 1: StockMarket s1,s2;
 2: s1= (StockMarket)
 3:     ServiceLocator.getRef("StockMarketA");
 4: s2= (StockMarket)
 5:     ServiceLocator.getRef("StockMarketB");
 6: ....
 7: StockInfo s;
 8: s=new StockInfo("sunw",124.6,"10/04/2007");
 9: s1.update(s);
10: .......
11: StockListener listener= new StockListenerImpl();
12: s1.subscribe("goog", listener);
13: s2.subscribe("yhoo", listener);
```

In lines 1 to 5, the client uses the `getRef` method to obtain references to the servers identified by the names `StockMarketA` and `StockMarketB` in the distribution descriptor presented in Figure 2. Next, the client calls the `update` method of the first server passing by-serialization an object of type `StockInfo` (lines 7 to 9). Finally, the client subscribes to changes in the price of two stocks (lines 11 to 13). It is worth to mention that in both calls the client informs the same object of type `StockListener` to receive callbacks from the servers. As prescribed by the associated distribution descriptor, the first server will use Java IDL to call back this object and the second server will use Java RMI.

3.3. SERVERS

DAJ generates for each server defined in a distribution descriptor an activation class, i.e., a class with a `main` method used to create, activate and register the remote object. This class has the same name of the server with the suffix `_Server`.

4. ARCHITECTURE

This section presents the software architecture and the implementation of the DAJ system, focusing on the abstract aspects defined by the system and the concrete aspects generated from the information defined in distribution descriptors. We begin the section describing the creation of remote interfaces. Next, we describe aspects used to inject middleware related concerns in business classes, retrieve remote references and activate remote objects. Thus, this section assumes that the reader is familiar with AspectJ syntax and semantics, which includes basic knowledge of abstractions such as join points, pointcuts, advices, and inter-type declarations. Such abstractions are described in details in AspectJ introductory papers [15] or textbooks [19].

4.1. REMOTE INTERFACES

Interfaces that have middleware related concerns tangled in their code are called remote interfaces. From distribution descriptor data, DAJ automatically generates remote interfaces as required by Java RMI and Java IDL. Such interfaces have the same name of their associated business interfaces, but they are created in a different package to avoid name collision.

Java RMI Remote Interfaces: Remote interfaces required by Java RMI are similar to their associated business interfaces, except that they must extend `java.rmi.Remote` and their methods must throw `java.rmi.RemoteException`. DAJ generates code for remote interfaces since the static crosscutting features of AspectJ do not support adding a `throws` clause in the signature of methods. This problem was already been reported in other works [29, 32].

Another difference between business and remote in-

terfaces is related to call by-remote-reference. Suppose that a business interface method has a parameter of a type `T` declared as remote in the distribution descriptor. In the remote interface, we must change `T` to the type of its associated remote interface. The reason is that Java RMI transmits the stub of the argument when the formal parameter implements a `Remote` interface. Thus, the stub is not compatible for assignment with business types.

Figure 3 presents the remote interface associated to the business interface `StockMarket`. The presented interface extends `Remote`. Moreover, its methods declare that they can throw `RemoteException`. Finally, the type of the parameter `obj` in the `subscribe` and `unsubscribe` methods was replaced by the associated remote interface type.

Java IDL Remote Interfaces: As usual in CORBA-based distributed applications, remote interfaces must be specified in IDL – a neutral language proposed by CORBA to define remote interfaces. Thus, DAJ generates an IDL specification for business types defined in the deployment descriptor of a given system. Moreover, DAJ relies on the `idlj` compiler that is part of the Java IDL platform to generate classes and interfaces required by CORBA implementations. In object-oriented Java IDL systems, the code of such classes is scattered and tangled in the business classes of the distributed system under design.

Figure 4 shows the interface `StockMarketOperations`, generated by the `idlj` compiler and that must be implemented by a business class. In this interface, business types have also been replaced by their counterpart remote types, as proposed by DAJ for RMI interfaces.

4.2. REMOTE CLASSES

Remote classes are those that result from the instrumentation of business classes with distribution concerns. DAJ makes a distinction between remote classes whose instances are registered in a naming service and those whose instances are not registered in such service. The former are always specified in a `server` tag of distribution descriptors. An example is the `StockMarketImpl` class. The latter are declared in `remote` tags and are used to create objects that are passed using call by-remote-reference in remote calls. An example is the class `StockListenerImpl`.

The remaining of this subsection describes how these two type of remote classes are instrumented with distribution code required by Java IDL. The instrumentation required by Java RMI follows the same pattern, and thus is not be presented in this paper.

Classes that can be located using a naming service: As any remote class, such classes must implement the remote interface generated by DAJ. Moreover, for each method in the remote interface with a parameter of type `T`, where `T` is a remote interface type, the system inserts a counterpart method in the class. The body of this method just calls the method that expects the business type associated with `T`.

In order to illustrate, we show the aspect that transforms `StockMarketImpl` in a remote class.

```
 1: aspect RemoteStockMarketImpl {
 2:   declare parents: StockMarketImpl implements
 3:                   StockMarketOperations;
 4:
 5:   public void StockMarketImpl.subscribe(
 6:            String stock,
 7:            stockwatcher.idl.StockListener obj) {
 8:     StockListenerAdapter adapter;
 9:     adapter= new StockListenerAdapter(obj);
10:     subscribe(stock, adapter);
11:   }
12:   ...
13: }
```

Lines 2-3 define that the class `StockMarketImpl` must implement the remote interface `StockMarketOperations` (described in Section 4.1). Since the `subscribe` method of this interface expects a remote interface type as parameter, a method with the expected signature is also introduced in the class `StockMarketImpl` (lines 5 to 11). This introduction is needed since the `subscribe` method originally available in `StockMarketImpl` expects a parameter of the type `StockListener` (instead of the remote type `stockwatcher.idl.StockListener` prescribed by the interface `StockMarketOperations`).

The method introduced in `StockMarketImpl` redirects the call to the original `subscribe` method of the class (line 10). In order to make the redirection possible the following adapter is employed:

```
class StockListenerAdapter implements StockListener {
  stockwatcher.idl.StockListener adaptee;
  StockListenerAdapter (
    stockwatcher.idl.StockListener adaptee) {
    this.adaptee= adaptee;
  }
  public void update(String stock, double value) {
    adaptee.update(stock,value);
  }
}
```

Essentially, this adapter conforms the `StockListener` remote interface to its counterpart business interface.

Classes that are not bound to a naming service: In such classes, the same inter-type declarations described previously are applied. Moreover, two new components are introduced by the associated aspect: a method `export2IDL` responsible for the activation of the remote object, and a reference `refIDL` to the stub created by the activation method provided by Java IDL. The

```
interface StockMarket extends Remote {
  void update(StockInfo info) throws RemoteException;
  void subscribe(String stock, stockwatcher.rmi.StockListener obj) throws RemoteException;
  void unsubscribe(String stock, stockwatcher.rmi.StockListener obj) throws RemoteException;
  StockInfo getStock(String stock) throws StockNotFoundException, RemoteException;
}
```

Figure 3. Java RMI interface for the motivating example

```
interface StockMarketOperations {
  void update(stockwatcher.idl.StockInfo info);
  void subscribe(String stock, stockwatcher.idl.StockListener obj);
  void unsubscribe(String stock, stockwatcher.idl.StockListener obj);
  stockwatcher.idl.StockInfo getStock(String stock)
     throws stockwatcher.idl.StockNotFoundException;
}
```

Figure 4. Java IDL interface for the motivating example

`export2IDL` method uses the value of `refIDL` to decide whether the object needs to be activated or not. If the value is `null`, the method activates the object and stores its remote reference in `refIDL`.

The following aspect is created by DAJ to transform the business class `StockListenerImpl` into a remote class.

```
 1: aspect RemoteStockListenerImpl {
 2:   declare parents: StockListenerImpl implements
 3:                   StockListenerOperations;
 4:
 5:   StockListener StockListenerImpl.refIDL= null;
 6:
 7:   StockListener StockListenerImpl.export2IDL() {
 8:     // activates remote object using Java IDL API
 9:   }
10: }
```

Line 3 defines that the class must implement the `StockListenerOperations` interface. The remainder of the aspect introduces in `StockListenerImpl` the `refIDL` field (line 5) and the `export2IDL` method (lines 7 to 9).

4.3. Obtaining Remote References

As defined in Section 3.2, clients obtain references to server objects by calling the `getRef` method. The implementation of this method basically retrieves the information about the requested object from the distribution descriptor and uses this information to perform a lookup operation in the naming server of Java RMI or Java IDL.

However, the `getRef` method does not return a reference to the stub of the remote object, but to a proxy for this object. This proxy is generated by DAJ and acts as representant for the remote object. The proxy implementation assumes the task of activating remote objects when they are passed as arguments of remote calls. In order to support this feature, the proxy relies on the `export2IDL` and `export2RMI` methods.

The following example presents the proxy returned when the client requests a reference to a server object of the type `StockMarketImpl`:

```
1: class StockMarketProxy implements StockMarket {
2:   stockwatcher.idl.StockMarket server;
3:   .....
4:   void subscribe(String stock, StockListener obj) {
5:     stockwatcher.idl.StockListener _obj;
6:     _obj= ((StockListenerImpl) obj).export2IDL();
7:     server.subscribe(stock, _obj)
8:   }....
9: }
```

The proxy `subscribe` method exports the object associated with its `obj` parameter (line 6) before redirecting the call to the remote object (line 7).

4.4. Activating Remote Servers

DAJ generates aspects that are responsible for creating, enabling and registering remote servers, thus removing this concern from business classes. Particularly for Java RMI, the following abstract aspect defines pointcuts and advices for handling these concerns:

```
1: abstract aspect RMIServer {
2:   abstract pointcut ServerMainExecution();
3:   abstract String getServerName();
4:   abstract String getRegistry();
5:   abstract Remote getInstance();
6:
7:   void around(): ServerMainExecution() {
8:       // code to create, activate, and register
9:       // remote objects
10:  }
11: }
```

In this aspect, the abstract pointcut `ServerMainExecution` includes the join points where remote servers should be created (line 2). The abstract method `getServerName` (line 3) returns a string that contains the name under which the object must be registered. The

abstract method `getRegistry` returns the host name of the naming server (line 4). The `getInstance` method is a factory method responsible for the creation of remote objects (line 5). The aspect also contains an `around` advice associated with the `ServerMainExecution` pointcut (lines 7 to 10). This advice contains code to create, activate, and register remote objects, by using methods provided by Java RMI.

DAJ also provides an abstract aspect called `CORBAServer`, that is similar to the aspect `RMIServer`.

Example: The concrete `RMIServerStocketMarketB` aspect created by DAJ activates and registers an instance of the `StockMarketB` server object defined in the distribution descriptor of the StockWatcher system.

```
aspect RMIServerStockMarketB extends RMIServer {
  pointcut ServerMainExecution: execution(
     public static void ServerStockMarketB.main(..));
  String getServerName() {
    return "StockMarketB";
  }
  String getRegistry() {
   return "patofu.pucminas.br:1530";
  }
  Remote getInstance() {
    return new StockMarketImpl();
  }
}
```

The information needed to generate this aspect – including the name of the remote object, the host name of the RMI Registry and the class of the remote object – was retrieved from the distribution descriptor of the StockWatcher system.

4.5. Exception Handling

This section describes how DAJ handles middleware concerns tangled in remote exception classes.

Exception handling in Java IDL: In Java IDL, exceptions must be specified in the IDL signature of remote methods. From IDL interfaces, the `idlj` tool generates classes that represent such exceptions in Java. However, the generated classes present tangled Java IDL code to handle the serialization of the exception, so that it can be propagated from the server to the client address space. For example, such classes must extend `org.omg.CORBA.UserException`.

On the other hand, the StockWatcher system also includes its own plain Java `StockNotFoundException` class:

```
class StockNotFoundException extends Exception {
   public StockNotFoundException(String msg) {
     super(msg);
   }
}
```

Since we do not want to create any explicit middleware interference in core classes, DAJ generates an aspect that handles remote exceptions in the client and server sides of distributed applications. This aspect includes an advice that catches business exceptions raised by methods of the `StockMarketImpl` class and throws an equivalent remote exception. The system then relies on the CORBA layer to propagate remote exceptions to the client side. Another advice catches remote exceptions when they arrive in the client proxy class. This advice then throws an equivalent business exception.

Exception handling in Java RMI: In Java RMI, business exceptions raised by remote methods are seamlessly propagated to client objects, using the Java built-in serialization mechanism. Therefore, there is no dichotomy between business and remote exception classes, as in Java IDL. However, Java RMI requires remote method signatures to declare a `RemoteException` checked exception, that is used to signal communication failures. In order to eliminate the need for handling such exception in the client code, DAJ generates an aspect that transforms it into an unchecked exception.

5. Experimental Study

This section describes the use of DAJ in the modularization of distribution concerns from three applications:

- HealthWatcher: a Web-based information system used by citizens to register complaints about the sanitary conditions of restaurants and food shops. A first experience on using AspectJ to modularize Java RMI concerns of HealthWatcher has been conducted by Soares, Borba and Laureano [30, 29]. In this paper, we have repeated this experience, but using DAJ instead of manually extracting aspects from HeathWatcher.
- Network Pricing System (NPS): a distributed system used to update and monitor stock prices. NPS has been originally designed as a text book example of a distributed application using many features provided by the Java IDL middleware platform [20].
- Library: a Java-RMI library management system with functions for handling customers, titles, copies, making reservations etc. Library has also been used to evaluate an instrumentation infrastructure for reverse engineering of UML sequence diagrams proposed by Briand, Labiche and Leduc [4].

In the study, we have initially refactored the original implementation of these systems, in order to remove distribution code from their business classes. Afterwards, we

have defined distribution descriptors for the evaluated systems. The tool `dajc` was then used to generate aspects including both Java RMI and Java IDL code.

The central idea of the study was to illustrate the ability of the system to handle distribution concerns from real-world applications. Since distribution is widely considered a crosscutting concern and thus a natural candidate for aspect-oriented approaches [30, 29, 32, 5, 21, 18], we only present information about the number of classes, interfaces and lines of code (LOC) regarding the original and refactored versions of the evaluated systems. Further studies may consider other object-oriented metrics in order to better demonstrate the advantages of using aspects for handling distribution.

5.1. HEALTHWATCHER

The object-oriented version of the HealthWatcher system relies on Java RMI for communication. Table 1 presents quantitative information about the system. HealthWatcher core has 78 classes and 13 interfaces. Distribution concerns, such as service lookup, remote exceptions and synchronization, are implemented by 4 classes and 2 interfaces. The main modularization drawback of this version of the system is the fact that code related to distribution is tangled in the classes and interfaces of the core.

	Classes	Interfaces	LOC
Core	78	13	4976
Distribution	4	2	153
Total	82	15	5129

Table 1. RMI-based HealthWatcher

DAJ-based HealthWatcher: Initially, a distribution descriptor was defined considering a deployment configuration employing two servers: one using Java RMI and the other relying on Java IDL. In this deployment descriptor, 25 classes designate objects passed by-serialization. HealthWatcher does not use call by-remote-reference.

Table 2 summarizes the components generated by the DAJ compiler from the HealthWatcher distribution descriptor. The new core of the system – after removing RMI classes and statements – has 410 lines less than the original core. Moreover, the original HealthWatcher has a very rigid architecture, including only one Java RMI server. On the other hand, the DAJ-based system presents a more flexible architecture that supports easily changing the underlying middleware.

It is worth to mention that Java IDL requires the generation of much more code than Java RMI (5671 lines for Java IDL and only 191 lines for Java RMI). The main reason is that the Java IDL compiler must generate several components that are mandatory in CORBA implementations, including stubs, portable adapters, classes responsible for IDL-to-Java mappings etc. The same difference in the number of lines between the two middleware systems will be observed in the following systems.

	C	I	A	LOC
Core	78	13	0	4566
DAJ internal components	6	0	0	244
Java RMI generated code	2	1	21	191
Java IDL generated code	103	29	38	5671
Total	189	43	59	10672

Table 2. DAJ-based HealthWatcher (C= Classes; I= Interfaces; A= Aspects)

5.2. NETWORK PRICING SYSTEM (NPS)

NPS functionality is very similar to the motivating example used in this paper. However, we choose to refactor NPS because it was originally implemented using Java IDL and uses many of the features provided by this middleware technology. Table 3 presents information about the original version of the system.

	Classes	Interfaces	LOC
Core	18	1	1279
Distribution	61	14	3138
Total	79	15	4417

Table 3. Java IDL-based NPS

DAJ-based NPS: First, Java IDL code was fully removed from NPS core. In this step, we also needed to incorporate new classes in the core in order to replace classes generated by the Java IDL compiler that have been used in the implementation of functional requirements of the system. This is the case, for example, of classes representing objects that are passed by-serialization in remote calls. Such classes are generated by the `idlj` tool and thus they have methods and fields required by Java IDL programming conventions. After replacing `idlj` generated classes by plain Java classes, the core was converted into an application that is independent from any middleware technology.

Table 4 summarizes the components generated by the DAJ compiler from the NPS distribution descriptor. As can be observed, the new core – considering the components added to replace Java IDL functional components – has 1210 lines, which is almost the same size of the original core. However, the new core does not depend anymore on any particular distribution technology. We were also able to change the communication middleware employed to access the two NPS server objects. In this way, it was very simple to build versions of NPS based on Java RMI only, on Java IDL only or on both platforms – for

example, one server can be configured to use Java RMI and the other one to use Java IDL.

	C	I	A	LOC
Core	25	8	0	1210
DAJ internal components	6	0	0	244
Java RMI generated code	7	7	18	355
Java IDL generated code	65	18	15	3744
Total	103	33	33	5553

Table 4. DAJ-based NPS (C= Classes; I= Interfaces; A= Aspects)

5.3. LIBRARY SYSTEM

The original version of the Library system uses Java RMI as the underlying communication infrastructure. Table 5 presents information about this version of the system. As can be observed in this table, the system does not have any class addressing distribution concerns (such as classes denoting remote exceptions). Basically, Library relies only on standard Java RMI classes that however are spreaded and tangled in the core functionality of the system.

	Classes	Interfaces	LOC
Core	66	4	4997

Table 5. Java RMI-based Library System

DAJ-based Library: First, we removed all RMI related concerns from the core. We had also to refactor remote method signatures when they include built-in classes from the Java API (such as `java.util.Vector`). The reason is that such classes are not supported by Java IDL – since Java IDL is a language neutral middleware platform. This restriction required us to implement new collection classes independent from the Java API.

The Library system has three remote servers. The first one handles functionality related to library employees. The second server handles tasks related to reservations, loans and payments. Finally, the last one is used by customers to search the library collection. Moreover, the system has 27 classes describing objects that must be passed in remote calls using call by-serialization. It does not make use of call by-remote-reference.

Table 6 summarizes the components generated by the `dajc` compiler from the Library distribution descriptor. It is worth to mention that the new core – including the new collection classes – has 193 more lines of code than the original one. However, this core is fully independent both from middleware technology and Java API components. Moreover, by just changing the middleware protocol in the distribution descriptor we were able to deploy a Java IDL version of the Library System.

	C	I	A	LOC
Core	70	5	0	5190
DAJ internal components	6	0	0	244
Java RMI generated code	6	3	26	309
Java IDL generated code	98	32	34	5427
Total	180	40	60	11170

Table 6. DAJ-based Libray (C= Classes; I= Interfaces; A= Aspects)

6. DISCUSSION

This section discusses the solution proposed by DAJ considering the following criteria: separation of concerns, usability and flexibility, portability, obliviousness and alternative implementation technologies.

Separation of Concerns: Using DAJ we were able to synthesize aspects and classes that modularize distribution concerns in the three evaluated systems. After refactoring, distribution concerns were completely removed from the core of these applications. However, we had to replace concrete components generated by Java IDL to components independent from any middleware technology. The same was required when remote method signatures rely on components from the Java API. A core independent from middleware concerns is more simple to understand, test and evolve. Particularly, tests can be performed without considering distribution, which is critical for example to the success of test-driven development.

Usability and Flexibility: Our experience demonstrates that defining distribution descriptors is fairly straightforward. It is very simple for example to change distribution parameters, such as the underlying middleware platform, naming servers location, remote object names etc. Moreover, it is also possible to reconfigure the distributed architecture of the base system. For example, we have rapidly reconfigured HealthWatcher to use two remote servers, instead of a single one. On the other hand, since it is based on XML, the distribution descriptort's syntax is not as legible and concise when compared with concrete syntax based on non-markup languages. Another drawback is that middleware parameters are hard-coded in distribution descriptors. Thus, DAJ does not provide support for example to applications that dynamically discover information about remote services. In the future, we have plans to tackle this problem. A possible solution is to extend the current API of the system, providing methods that allow clients to set up middleware parameters at run-time.

Performance: In order to evaluate the performance of the system, we have measured the time to execute the remote

calls included in the StockWatcher system. This system was executed 20 times. In each execution, each remote call of the system was dispatched ten thousand times. We then calculated the average time to perform the dispatched calls considering the sequences of 20 executions. Moreover, we have evaluated four versions of StockWatcher: based on Java RMI (using DAJ and using object-oriented code) and on Java IDL (using DAJ and object-oriented code). The results are presented on Tables 7 and 8. We run client and server processes on a Pentium 4 3.2 GHz, 1 GB RAM, Microsoft Windows XP Service Pack 2, JDK 1.6 and `ajc` version 1.5.4.c.

The experiment demonstrates that DAJ does not impact significantly middleware performance. The performance overhead is negligible in most of the measured calls, both for Java RMI and Java IDL versions of the system. The only exception is when the `getStock` method raises a `StockNotFoundException` in the Java IDL-based version of StockWatcher. The performance overhead in this case is higher than 8%, compared with the original implementation of the system using scattered and tangled OO code. This overhead is due to the need of using aspects to handle such exception both on the server and on the client side of the distributed system, as described in Section 4.5.

Portability: Using DAJ, we were able to deliver versions of the evaluated systems for a different middleware platform than the one supported by their original implementation. However, DAJ only provides support for the "lowest common denominator" between Java IDL and Java RMI. Basically, this translates to synchronous remote calls using call by-serialization and call by-remote-reference semantics. DAJ does not provide support for example to features available only in CORBA systems, such as asynchronous calls, oneway calls, portable interceptors, dynamic invocation interfaces etc. On the other hand, we envision that the system can be extended to support other CORBA ORBs, besides Java IDL.

Another challenge is to provide support for Web Services in DAJ. In fact, we have already investigated this issue, considering Apache Axis as the middleware platform for supporting Web Services [2]. Basically, we faced two problems. First, Apache Axis generates its own classes for types used in remote service invocations. Such classes contain methods that serializes and deserializes objects according to Web Services standards. Thus, supporting Web Services in DAJ would require several inter-type declarations to transpose such methods to the classes of the base system. Another challenge is that Web Services do not support call by-remote-reference, since this would require the deployment of an HTTP server in each Web Service client. For this reason, we decided to do not support Web Services in the first DAJ implementation.

In summary, we consider that DAJ is a solution that targets only crosscutting concerns inherent to distributed object architectures. Particularly, we do not expect that the proposed system can handle concerns related to other middleware paradigms, such as service-oriented, message-oriented or event-based.

In the future, we consider that DAJ can be extended to generate code for other programming languages, such as C++ or C#. The only requirement is that aspect extensions for such OO languages become more mature.

Obliviousness: Although DAJ fully insulates middleware code from application components, it is worth to mention that DAJ requires client modules to rely on its own `getRef` method to retrieve remote references. Thus, it can be argued that DAJ replaces middleware tangling code by its own. However, this degree of tangling is minimal and it can be eliminated by creating an extra aspect, as illustrated below for an hypothetical StockWatcher client:

```
 1: class MyClient {
 2:   StockMarket s1,s2;
 3:   ....
 4: }
 5: aspect MyClientDependencyInjection {
 6:  after(MyClient c):
 7:   execution(void MyClient.new(..)) && this(s) {
 8:     c.s1= (StockMarket)
 9:           ServiceLocator.getRef("StockMarketA");
10:     c.s2= (StockMarket)
11           ServiceLocator.getRef("StockMarketB");
12:   }
13: }
```

The proposed aspect, as happen in dependency injection frameworks [24, 6], assumes the charge of retrieving and binding remote references to instance variables `s1` and `s2` (lines 8 to 11). In this way, the class has remained oblivious to DAJ concerns.

Alternative Technologies: Instead of aspects, at least two other technologies could have been used to modularize distributions concerns required by middleware platforms:

- Bytecode manipulation frameworks: Frameworks such as BCEL [9] could have been used to instrument the bytecode of the core classes with middleware related programming conventions. However, aspect-oriented languages, such as AspectJ, provide high-level abstractions to perform the same kind of instrumentation. Such abstractions have contributed to simplify the design and implementation of DAJ.

- Model-based Development: As promoted by model-driven approaches [25], CASE tools can be augmented to support the generation of class skeletons incorporating middleware required code. Such

Remote Call	RMI-DAJ	RMI-OO	DAJ / OO (%)
`update`	2767	2698	2.56
`subscribe`	5901	5803	1.69
`unsubscribe`	3010	2989	0.70
`getStock` returning `StockInfo`	2631	2648	-0.64
`getStock` returning an exception	5089	5045	0.87

Table 7. Average time (in ms) to perform ten thousand remote calls in the Java RMI-based versions of the StockWatcher system (using DAJ and using tangled and spreaded OO code)

Remote Call	IDL-DAJ	IDL-OO	DAJ / OO (%)
`update`	6473	6450	0.36
`subscribe`	13809	13634	1.28
`unsubscribe`	6428	6289	2.21
`getStock` returning `StockInfo`	6326	6332	-0.09
`getStock` returning an exception	6401	5893	8.62

Table 8. Average time (in ms) to perform ten thousand remote calls in the Java IDL-based versions of the StockWatcher systems (using DAJ and using tangled and spreaded OO code)

classes can be generated from models, such as UML class and sequence diagrams, or from domain-specific languages. However, this approach usually generates code that is hard to understand and evolve. Moreover, since application logic is embedded in the synthesized class skeletons, it is usually hard to achieve round-trip engineering, i.e., moving back and forth between models and generated code [13].

7. Related Work

Soares, Borba and Laureano have reported their experience using AspectJ to provide an aspect-oriented implementation for the HealthWatcher system [30, 29]. Regarding their work, DAJ presents at least three contributions:

- DAJ supports modularization both for Java RMI and Java IDL communication concerns. Supporting Java IDL is important to provide interoperability with clients and servers written in other programming languages. The challenge in this case was to avoid collisions and interferences among aspects specific to each middleware system. For example, when the aspects proposed by DAJ require inter-type declaration, interface implementation was preferred since Java only supports simple inheritance.

- DAJ supports parameter passing in remote calls using call by-serialization and call by-remote-reference semantics. Instead, the aspects proposed by Soares, Borba and Laureano only handle call by-serialization, since this was the only evaluation strategy employed in HealthWatcher. However, call by-remote-reference is extensively used in other distributed object-based systems to support the implementation of callbacks [11].

- DAJ is compatible with different software architectures employed in the design of distributed systems. On the other hand, the solution proposed by Soares and colleagues is restricted to the original architecture employed by the HealthWatcher system. For example, their solution assumes that there is a single server object of type `HWFacade`. The proposed pointcuts rely on this particular type to capture remote calls. For this reason, aspects used in HealthWatcher do not apply for example to clients that reference multiple server objects.

Kulesza et al. have conducted a quantitative study to assess the positive and negative effects of using AOP in the HealthWatcher system [18]. Their study was driven by a suite of metrics for separation of concerns, coupling, cohesion and size. The conclusion is that the AO version of the system presents better results for almost all the considered metrics. Other works also report the advantages of using aspects to modularize distribution concerns [32, 5, 21].

Ceccato and Tonella have proposed an aspect-oriented framework to support the migration of a non-distributed application to a distributed architecture [5]. Similar to DAJ, the object-oriented code remains oblivious to the injected distributed behavior and the proposed aspects are generated automatically. However, their solution is restricted to Java RMI. Moreover, the configuration file that guides the generation of aspects is just a list of class names, which does not contain information about parameter passing strategies, naming servers, etc. In their so-

lution, call by-remote-reference is applied to all invocation parameters. Certainly, this decision is not adequated when the target of the modularization is an existing distributed application whose semantics depends on call by-serialization. Additionally, using only call by-remote-reference impacts the performance of the system, since the execution of remote methods must call back the client address space to access any argument.

Reflective middleware systems rely on the application of reflection to achieve customizable, open, and adaptive middleware platforms [17]. For example, UIC CORBA supports the concept of personality, that allows users to specialize middleware systems according to the requirements of a given distributed application or domain [28]. It is possible for example to configure skeletons that support communication with different middleware systems. Pluggable protocols aim to support custom middleware protocol stacks in TAO, a well-known CORBA middleware framework [27]. The system targets mainly applications with sensitive time constraints, such as real-time and embedded systems. However, as usual in object-oriented middleware systems, both UIC and TAO require developers to follow their specific programming conventions, which usually lead to code scattering and tangling [22].

Ghosh and colleagues have proposed a middleware transparent approach to support the implementation of distributed systems [12]. Similar to DAJ, they rely on aspects to decouple business concerns from middleware-specific functionality. Moreover, they propose a model-driven approach to distributed systems development. A middleware transparent design (MTD) is a model of the system that do not address middleware concerns. Aspects are weaved to such model to derive a middleware specific design (MSD) that addresses both business and distribution concerns. Accordingly to Model Driven Architecture (MDA) principles [25], a MTD corresponds to a PIM (Platform Independent Model) and a MSD to a PSM (Platform Specific Model). DAJ is a system that implements and puts into practice many principles and ideas proposed by Ghosh and colleagues in their proposal.

Zhang and Jacobsen have quantified the crosscutting nature of several features of CORBA based middleware [37]. They have measured the scattering degree of features such as portable interceptors, dynamic programming invocations, collocation optimizations, and asynchronous calls. They have also showed that such features can be modularized using aspect-oriented programming. From this experience, they have proposed the horizontal decomposition method [38]. Horizontal decomposition advocates the use of traditional modularization techniques, such as vertical decomposition, to implement a minimal but well-modularized core middleware system. Aspects should then be used to superimpose orthogonal features to this core. They have assessed the effectiveness of their method by re-implementing as aspects crosscutting features of the original implementation of ORBacus [26]. AspectJRMI is another middleware system whose design has been guided by horizontal decomposition principles [10]. Colyer et al. have conducted an aspect-oriented refactoring of a large scale middleware product-line [7]. They have identified many crosscutting concerns in the product-line, including homogenous concerns (tracing, logging, error analysis and reporting, monitoring, and statistics) and heterogeneous concerns (EJB support). Similar to Zhang and Jacobsen, Colyer work aims to modularize crosscutting concerns that are internal to the middleware layer.

In DAJ-based distributed systems, the language employed in the specification of distribution descriptors can be considered as a domain-specific aspect-oriented language. Recently, some works have proposed the use of domain-specific languages to describe crosscutting concerns, since such languages rely on abstractions and vocabulary that is closer to the concern they intend to modularize. Moreover, they are less susceptible to critics commonly related to general purpose aspect-oriented languages. For example, many researchers consider that languages such as AspectJ defeats basic software engineering principles, such as modular reasoning, parallel development and encapsulation [35, 31]. Certainly, such critics are less important when using domain-specific aspect languages. It is also worth to mention that the first generation of aspect languages was defined for domain-specific purposes, including RIDL (for remote interface specification) and COOL (for coordination and synchronization) [21].

8. CONCLUSIONS

In this paper, we have described a system called DAJ that provides middleware independence in distributed, object-oriented systems. For this purpose, DAJ relies on the combination of three technologies: aspects, domain-specific languages and generative programming. DAJ also supports two native, object-oriented JDK middleware platforms: Java RMI and Java IDL. The system modularizes the basic service provided by these middleware platforms, i.e. synchronous remote calls using call by-serialization and call by-remote-reference semantics. Particularly, DAJ does not provide support to features available only in CORBA systems, such as asynchronous calls, oneway calls, portable interceptors, dynamic invocation interfaces etc.

In order to evaluate the proposed system, we have refactored three medium-sized distributed applications: HealthWatcher, Network Pricing System, and Library. DAJ was able to modularize middleware code tangled in the business components of these three systems. We have

also been able to deploy a new version of the evaluated systems using a different middleware platform than the original one.

As future work, we have plans to investigate the use of DAJ in other distributed systems, including applications designed to rely on DAJ since the design phase. We also have plans to consider other object-oriented metrics in the investigated systems, including metrics that evaluate important software engineering attributes, such as cohesion, coupling and separation of concerns.

Acknowledgments This work has been funded by FAPEMIG (process CEX-817/05). We would like to thank Sergio Soares for providing the source code of the HealthWatcher system and Mariana Sharp for providing the Library system source code.

REFERENCES

[1] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architecture and Applications*. Springer Verlag, 2004.

[2] Apache Axis. http://ws.apache.org/axis/.

[3] Jon Bentley. Programming pearls: little languages. *Communications ACM*, 29(8):711–721, 1986.

[4] Lionel C. Briand, Yvan Labiche, and Johanne Leduc. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Transactions on Software Engineering*, 32(9):642–663, 2006.

[5] Mariano Ceccato and Paolo Tonella. Adding distribution to existing applications by means of aspect oriented programming. In *4th IEEE International Workshop on Source Code Analysis and Manipulation*, pages 107–116. IEEE Computer Society, 2004.

[6] Shigeru Chiba and Rei Ishikawa. Aspect-oriented programming beyond dependency injection. In *19th European Conference on Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 121–143. Springer, 2005.

[7] Adrian Colyer and Andrew Clement. Large-scale AOSD for middleware. In *3rd International Conference on Aspect-Oriented Software Development*, pages 56–65. ACM Press, 2004.

[8] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[9] Markus Dahm. Bytecode engeeniering with the BCEL API. Technical report, Freie Universitt - Institut fur Informatik, 2001.

[10] Marco Tulio de Oliveira Valente, Fabio Tirelo, Diana Campos Leao, and Rodrigo Palhares. An aspect-oriented communication middleware system. In *International Symposium on Distributed Objects and Applications*, volume 3761 of *LNCS*, pages 1115–1132. Springer-Verlag, October 2005.

[11] Frantisek Plasil and Michael Stal. An architectural view of distributed objects and components in CORBA, Java RMI and COM/DCOM. *Software Concepts and Tools*, 19(1):14–28, 1998.

[12] Sudipto Ghosh, Robert B. France, Devon M. Simmonds, Abhijit Bare, Brahmila Kamalakar, Roopashree P. Shankar, Gagan Tandon, Peter Vile, and Shuxin Yin. A middleware transparent approach to developing distributed applications. *Software Practice and Experience*, 35(12):1131–1154, October 2005.

[13] Aniruddha S. Gokhale, Douglas C. Schmidt, Balachandran Natarajan, and Nanbor Wang. Applying model-integrated computing to component middleware and enterprise applications. *Communications of the ACM*, 45(10):65–70, 2002.

[14] Java IDL. http://java.sun.com/products/jdk/idl.

[15] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *15th European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *LNCS*, pages 327–355. Springer Verlag, 2001.

[16] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[17] Fabio Kon, Fábio Costa, Roy Campbell, and Gordon Blair. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, June 2002.

[18] Uirá Kulesza, Cláudio Sant'Anna, Alessandro Garcia, Roberta Coelho, Arndt von Staa, and Carlos José Pereira de Lucena. Quantifying the effects of aspect-oriented programming: A maintenance study. In *22nd IEEE International Conference on Software Maintenance*, pages 223–233, 2006.

[19] Ramnivas Laddad. *AspectJ in Action Practical Aspect-Oriented Programming*. Manning, 2003.

[20] Geoffrey Lewis, Steven Barber, and Ellen Siegel. *Programming with Java IDL*. John Wiley & Sons, 1997.

[21] Cristina Videira Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, December 1997.

[22] Neil Loughran, Lionel Seinturier Geoff Coulson, Renaud Pawlak, Eddy Truyen, Frans Sanen, Maarten Bynens, Wouter Joosen, Andrew Jackson, Siobhan Clarke, Neil Hatton, Monica Pinto, Lidia Fuentes, Mercedes Amor, Tal Cohen, Adrian Colyer, and Christa Schwanninger. Requirements and definition of aspect-oriented middleware reference architecture. Technical report, AOSD Europe, oct 2005.

[23] Cristiano Amaral Maffort and Marco Tulio de Oliveira Valente. Aspectos para construção de aplicações distribuídas. In *XX Simpósio Brasileiro de Engenharia de Software*, pages 271–286, 2006.

[24] Martin Fowler. Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html.

[25] OMG Model Driven Architecture. http://www.omg.org/mda.

[26] Orbacus. http://www.orbacus.com.

[27] Carlos O'Ryan, Fred Kuhns, Douglas C. Schmidt, Ossama Othman, and Jeff Parsons. The design and performance of a pluggable protocols framework for real-time distributed object computing middleware. In *International Middleware Conference*, volume 1795 of *Lecture Notes in Computer Science*, pages 372–395. Springer, 2000.

[28] Manuel Román, Fabio Kon, and Roy Campbell. Reflective middleware: From your desk to your hand. *Distributed Systems Online*, 2(5), July 2001.

[29] Sergio Soares, Paulo Borba, and Eduardo Laureano. Distribution and persistence as aspects. *Software Practice and Experience*, 36(7):711–759, 2006.

[30] Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *17th ACM Conference on Object-Oriented programming systems, languages, and applications*, pages 174–190. ACM Press, 2002.

[31] Friedrich Steimann. The paradoxical success of aspect-oriented programming. In *21st Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 481–497, 2006.

[32] Eli Tilevich, Stephan Urbanski, Yannis Smaragdakis, and Marc Fleury. Aspectizing server-side distribution. In *Automated Software Engineering Conference*, pages 130–141. IEEE Press, October 2003.

[33] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[34] Steve Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), 1997.

[35] Mitchell Wand. Understanding aspects: extended abstract. In *8th International Conference on Functional Programming*, pages 299–300. ACM Press, August 2003.

[36] Ann Wollrath, Roger Riggs, and Jim Waldo:. A distributed object model for the Java system. In *2nd Conference on Object-Oriented Technologies & Systems*, pages 219–232, 1996.

[37] Charles Zhang and Hans-Arno Jacobsen. Refactoring middleware with aspects. *IEEE Transactions Parallel and Distributed Systems*, 14(11):1058–1073, 2003.

[38] Charles Zhang and Hans-Arno Jacobsen. Resolving feature convolution in middleware systems. In *19th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 188–205. ACM Press, 2004.