Aspect-Oriented Programming Beyond Dependency Injection

Shigeru Chiba and Rei Ishikawa

Dept. of Mathematical and Computing Sciences,
Tokyo Institute of Technology

Abstract. Dependency injection is a hot topic among industrial developers using component frameworks. This paper first mentions that dependency injection and aspect-oriented programming share the same goal, which is to reduce dependency among components for better reusability. However, existing aspect-oriented programming languages/ frameworks, in particular, AspectJ, are not perfectly suitable for expressing inter-component dependency with a simple and straightforward representation. Their limited kinds of implicit construction of aspect instances (or implementations) cannot fully express inter-component dependency. This paper points out this fact and proposes our aspect-oriented programming system named *GluonJ* to address this problem. GluonJ allows developers to explicitly construct and associate an aspect implementation with aspect targets.

1 Introduction

A key feature of the new generation of component frameworks like the Spring framework [10] is dependency injection [6]. It is a programming technique for reducing the dependency among components and thereby improving the reusability of the components. If a component includes sub-components, reusing only that component *as is* independently of those sub-components is often difficult. For example, if one of those sub-components is for accessing a particular database, it might need to be replaced with another sub-component for a different database when the component is reused. The original program of that component must be edited for the reuse since it includes the code for instantiating the sub-component. The idea of dependency injection is to move the code for instantiating sub-components from the program of a component to a component framework, which makes instances of sub-components specified by a separate configuration file (usually an XML file) and automatically stores them in the component.

Dependency injection is a good idea for reducing inter-component dependency. However, since existing component frameworks with dependency injection are implemented with a normal language, mostly in Java, the independence and reusability of components are unsatisfactory. For example, the programs of components depend on a particular component framework and thus they must be modified when they are reused with a different framework.

A.P. Black (Ed.): ECOOP 2005, LNCS 3586, pp. 121–143, 2005.

This paper mentions that dependency injection and aspect-oriented programming (AOP) share the same goal from a practical viewpoint. Hence introducing the ideas of aspect-oriented programming into this problem domain provides us with better ability for reducing dependency among components. However, existing aspect-oriented systems used with component frameworks are mostly based on the architecture of AspectJ [12] and thus their design has never been perfectly appropriate for reducing inter-component dependency. In fact, aspect-oriented programming and dependency injection have been regarded as being orthogonal and used for different applications and purposes. Otherwise, aspect-oriented programming is just an implementation mechanism of dependency injection.

This paper presents our aspect-oriented programming framework named *GluonJ*, which we designed for dealing with dependency among components in Java. Although the basic design of GluonJ is based on that of AspectJ, GluonJ allows developers to explicitly associate an aspect implementation with aspect targets. The aspect implementation is a component implementing a crosscutting concern and the aspect targets are components that the concern cuts across. Existing aspect-oriented systems only allow implicit association and hence they cannot fully express inter-component dependency as an aspect.

The organization of the rest of this paper is followings. In Section 2, we discuss dependency injection and problems of the current design. Section 3 presents our aspect-oriented programming framework named GluonJ. Section 4 mentions comparison between GluonJ and AspectJ. Section 5 briefly describes related work and Section 6 concludes this paper.

2 Loosely Coupled Components

This section first overviews the idea of dependency injection. Then it mentions that dependency injection makes components dependent on a particular component framework and a naive aspect-oriented solution is not satisfactory.

2.1 Dependency Injection

Dependency injection enables loosely-coupled components, which are thereby highly reusable. If a component contains a sub-component, it will be usually difficult to reuse without the sub-component since these two components will be tightly coupled. For example, suppose that the program of that component is as following (Figure 1):

```
public class MyBusinessTask {
  Database db;

  public MyBusinessTask() {
    db = new MySQL();
  }
```

```
  public void doMyJob() {
    Object result = db.query("SELECT USER.NAME FROM USER");
    System.out.println(result);
  }
}
```

Note that this component contains a MySQL object as a sub-component. MySQL is a class implementing a Database interface. Since MyBusinessTask is tightly coupled with MySQL, the constructor of MyBusinessTask must be modified if MyBusinessTask is reused with another database accessor, for example, a PostgreSQL object. The new constructor would be:

```
public MyBusinessTask() {
  db = new PostgreSQL();    // not new MySQL()
}
```

Dependency injection loosens the connection between MyBusinessTask and MySQL. It enables us to reuse MyBusinessTask without modification even if we must switch a database accessor from MySQL to PostgreSQL. The program of MyBusinessTask would be changed into this:

```
public class MyBusinessTask {
  Database db;

  public void setDb(Database d) {
    db = d;
  }

  public void doMyJob() {
    Object result = db.query("SELECT USER.NAME FROM USER");
    System.out.println(result);
  }
}
```

Now, no constructor in MyBusinessTask initializes the value of the db field. It is initialized (*or injected*) by a factory method provided by the framework

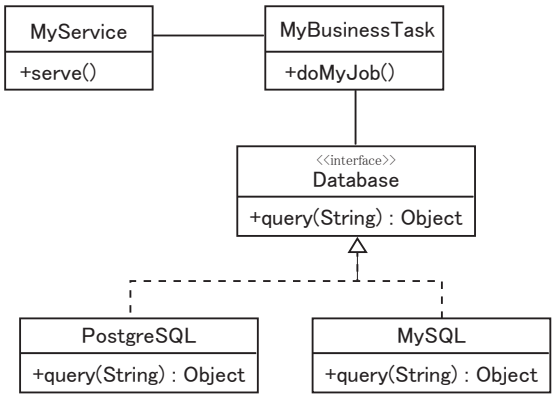

Fig. 1. Class diagram for our example scenario

supporting dependency injection. Thus, a MyBusinessTask object must not be constructed by the new operator but the factory method (or, otherwise, a MyBusinessTask object constructed by the new operator must be explicitly passed to a method provided by the component framework for performing dependency injection). For example, the code snippet below constructs a MyBusinessTask object:

```
XmlBeanFactory factory = new XmlBeanFactory(
    new InputStreamResource(new FileInputStream("beans.xml")));
MyBusinessTask myTask = (MyBusinessTask)factory.getBean("myTask");
```

Here, XmlBeanFactory is a factory class provided by a component framework. The getBean method constructs an instance of MyBusinessTask and initializes the value of the db field. It constructs a MySQL object and assigns it to the db field. This initialization is executed according to an XML configuration file beans.xml. The parameter to getBean is a key to find a configuration entry for MyBusinessTask in beans.xml.

Reusing MyBusinessTask with not MySQL but PostgreSQL is easy. We do not have to modify the program of MyBusinessTask but we have only to modify the configuration file beans.xml, which specifies how the db field is initialized. According to the configuration file, the getBean method will construct a PostgreSQL object and assign it to the db field.

However, using a factory method is annoying. Furthermore, if the hierarchy of components is more complicated, the program of the components depends on a particular component framework. Suppose that MyBusinessTask is a subcomponent of another component MyService. The program of MyService would be something like this:

```
public class MyService {
  MyBusinessTask task;

  public MyService(XmlBeanFactory factory) {
    task = factory.getBean("myTask");
  }

  public void serve() {
    task.doMyJob();
  }
}
```

MyService and MyBusinessTask do not require to be modified when they are reused with either MySQL or PostgreSQL. Only the configuration file must be modified.

However, the program above includes XmlBeanFactory, which is a class provided by the component framework. MyService and MyBusinessTask, therefore, depend on the component framework. We cannot reuse them *as is* with another component framework. If we switch component frameworks, we also have to modify the MyService class. This problem can be avoided if we also construct a MyService object through an XmlBeanFactory object. Since the component

framework constructs a MyBusinessTask object for injecting it into the task field in a MyService object, the constructor of MyService does not have to explicitly call the getBean method on factory. We can write the program of MyService without referring to XmlBeanFactory. However, this solution requires all components to be constructed through an XmlBeanFactory object. For example, MyService might be always reused with MyBusinessTask since these two components are tightly coupled. If so, dependency injection is not necessary for MyService but we must write a configuration file for MyService and construct the MyBusinessTask component through an XmlBeanFactory object. This programming convention is somewhat awkward.

2.2 Aspect-Oriented Programming

The programming problem of dependency injection mentioned above is that the programs of components depend on a particular component framework; we cannot switch component frameworks without modifying the programs of the components. We must stay with a particular component framework. As we develop a larger collection of useful components, switching component frameworks becomes more difficult.

This problem can be easily solved if we accept aspect-oriented programming. Since the source of this problem is that we cannot intercept object construction within confines of regular Java, we can solve the problem by using aspect-oriented programming for intercepting object creation. For example, if we use AspectJ, we can intercept construction of MyBusinessTask by the following program:

```
privileged aspect DependencyInjection {
  after(MyBusinessTask s):
      execution(void MyBusinessTask.new(..)) && this(s) {
    s.db = new MySQL();
  }
}
```

This aspect corresponds to an XML configuration file of the component framework shown in the previous subsection. If we define this aspect, the definition of MyService can be written without a factory method:

```
public class MyService {
  MyBusinessTask task;

  public MyService() {
    task = new MyBusinessTask();
  }

  public void serve() {
    task.doMyJob();
  }
}
```

If this class is compiled with the aspect, the construction of MyBusinessTask is intercepted and then a MySQL object is assigned to the db field in MyBusinessTask.

Although this solution using AspectJ requires our development environments to support a new language — AspectJ, we can use a Java-based aspect-oriented framework such as JBoss AOP[9] and AspectWerkz [1] if we want to stay with regular Java.

The solution with aspect-oriented programming enables reusable components that are even independent of component frameworks. We can switch component frameworks and aspect-oriented programming systems without modifying the definition of MyService. Only the DependencyInjection aspect must be rewritten if the aspect-oriented programming system is changed.

2.3 Is This Really a Right Solution?

Although AspectJ could make a MyService component independent of a component framework, this solution would not be a good example of aspect-oriented programming. This solution uses AspectJ only as an implementation mechanism for intercepting object construction. It can be implemented with not only AspectJ but another mechanism such as a metaobject protocol [11, 3, 7, 20], which also enables intercepting object construction. In fact, the description in the DependencyInjection aspect does not directly express the dependency relation among components. It is procedural and includes implementation details. The level of abstraction is relatively low.

However, we mention that aspect-oriented programming is a right approach to solve the problem illustrated above, and more generally, to reduce dependency among components. In other words, aspect-oriented programming and dependency injection share the same goal, which is to reduce inter-component dependency for better component reusability. Aspect-oriented programming is known as a paradigm for implementing a crosscutting concern as an independent and separated component. A concern is called crosscutting if the implementation of that concern in a non aspect-oriented language is tangled with the implementation of other components. Another interpretation of this definition is that aspect-oriented programming is a paradigm for separating tightly coupled components so that they will be less dependent on each other and hence easily reusable. This is the same goal of dependency injection although dependency injection can reduce only a particular kind of dependency while aspect-oriented programming covers a wider range of dependency.

Unfortunately, existing aspect-oriented programming systems represented by AspectJ are not perfectly suitable to reduce inter-component dependency. A main problem is that they implicitly associate an aspect implementation with an aspect target. Here, the aspect implementation is a component implementing a crosscutting concern and the aspect target is a component that the concern cuts across. If program execution reaches a join point specified by a pointcut, an advice body is executed on the aspect implementation associated with the aspect target including that join point. In AspectJ, an aspect implementation is not a regular Java object but an instance of aspect. Thus we cannot use an existing component written in Java as an aspect implementation. To avoid this problem, several frameworks such as JBoss AOP and AspectWerkz allow

using a regular object as an aspect implementation. Pointcuts are described in an aspect-binding file, that is, an XML file. However, such a regular object is *implicitly* constructed and associated with the aspect target. An aspect instance of AspectJ is also implicitly constructed and associated.

This implicit association between an aspect implementation and an aspect target has two problems. First, expressing dependency injection is made difficult. Dependency injection can be regarded as associating a component with another. An injected component corresponds to an aspect implementation. If developers do not have full control of the association, they cannot naturally express dependency injection with aspect-oriented programming.

The other problem is that the implicit association does not provide sufficient expressive power enough to express various relations of inter-component dependency as aspects. Although AspectJ lets developers select a scheme from issingleton, perthis, and so on, these options cover only limited kinds of relations among components. Developers might want to associate an aspect implementation with a group of aspect targets. AspectJ does not support this kinds of association. The relations among components in general do not always form a simple tree structure. Hence an aspect implementation is not always a sub-component owned by a single aspect target. It may be referred to by several different aspect targets. It may be a singleton and hence shared among all aspect targets. Existing aspect-oriented programming systems allow only selecting from limited types of the relation and they implicitly construct an aspect implementation and associate it with the aspect target according to the selected option. Therefore, developers must often redesign the relations among components so that the relations fit one of the types provided by the system.

3 GluonJ

This section presents *GluonJ*, which is our new aspect-oriented programming framework for Java. The design of GluonJ is based on the pointcut-advice architecture of AspectJ. However, this architecture has been restructured for GluonJ to provide a simpler programming model for reducing inter-component dependency.

GluonJ separates aspect bindings from aspect implementations. Aspect implementations are regular Java objects, which implement a crosscutting concern. They corresponds to an aspect instance in AspectJ. Aspect binding is the *glue* code described in XML. It specifies how an aspect implementation is associated with aspect targets, which the aspect implementation cuts across, at specified join points. The aspect binding includes not only pointcuts but also code fragments written in Java. These code fragments *explicitly* specify which aspect implementation is associated with the aspect targets. If program execution reaches a join point specified by a pointcut, then the code fragment is executed. It can explicitly construct an aspect implementation and call a method on that aspect implementation to execute a crosscutting concern. Since GluonJ was designed for reducing inter-component dependency, GluonJ lets developers to describe

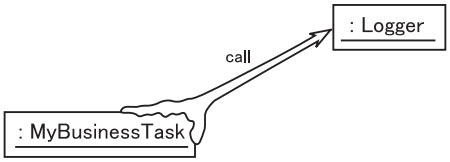

Fig. 2. The aspect of GluonJ is *glue*, which connects two components. Unlike the aspect of AspectJ, the aspect of GluonJ is not part of the Logger component or the MyBusinessTask component

the code fragments in the aspect binding to explicitly express various relations between aspect targets and aspect implementations.

3.1 Logging Example

To illustrate the usage of GluonJ, we below show the implementation of a logging concern in GluonJ. The logging concern is a well-known crosscutting concern, which is often used for showing the usage of an aspect-oriented programming system. The goal of this example is to extend the behavior of MyBusinessTask so that a log message will be printed when a method in MyBusinessTask is executed. However, we cannot modify the program of MyBusinessTask for this extension since modifying that program means that MyBusinessTask includes part of the implementation of the logging concern. The logging concern must be implemented as an independent component separated from the other components.

In GluonJ, we first define a Logger class in Java:

```
public class Logger {
  public void log() {
    System.out.println("method execution");
  }
}
```

Logger is a class for the logging concern. Unlike AspectJ, GluonJ uses a regular Java object as an aspect implementation, which is a component implementing a crosscutting concern such as the logging concern.

In GluonJ, an aspect means the aspect binding written in XML, for example, for describing the dependency between a Logger object and other objects. It glues a Logger object to the objects that must produce log messages (Figure 2). The aspect does not include an aspect implementation, which is the Logger class. For example, the following aspect specifies that a log message is printed just after a method in MyBusinessTask is executed:

```
<aspect>
  <injection>
    Logger MyBusinessTask.aspect = new Logger();
  </injection>
  <advice>
    <pointcut>
```

```
      execution(* MyBusinessTask.*(..))
    </pointcut>
    <after>
      Logger.aspectOf(this).log();
    </after>
  </advice>
</aspect>
```

This aspect makes it possible to keep the two components MyBusinessTask and Logger loosely coupled with low dependency on each other. The GluonJ compiler automatically transforms the program of MyBusinessTask according to this aspect at compilation time. Thus, we can change the behavior of MyBusinessTask without manually modifying the program of MyBusinessTask.

The statement surrounded with the injection tag specifies a connection between a MyBusinessTask object and a Logger object. It means that, when a MyBusinessTask is constructed, a Logger object is also constructed and then associated with that MyBusinessTask object. The syntax of this statement is the same as the intertype field declaration in AspectJ except aspect is not a field name but a special keyword.

The elements surrounded with the advice tag are pointcut and after advice. The pointcut is surrounded with the pointcut tag. It is the almost same language element as AspectJ's pointcut except the syntax. In the aspect shown above, the pointcut picks out as join points method execution on MyBusinessTask objects. The code snippet surrounded with the after tag is an advice body, which is executed just after a thread of control reaches the execution point specified by the pointcut. The code snippet is written in regular Java except that a special form aspectOf is available in that code snippet. In the aspect shown above, Logger.aspectOf(this) is used to obtain the Logger object associated with the MyBusinessTask object referred to by this. aspectOf is a special form that is used in the following form:

<*class name*>`.aspectOf(`<*object*>`)`

This special form is used to obtain an object associated with another object by the injection tag. It returns the object that is of the <*class name*> type and is associated with the given <*object*>.

The advice body, which is the code snippet surrounded with the after tag, is executed in the context of the join point picked out by a pointcut. In the case of our example, the advice body is executed on an MyBusinessTask object since the join points picked out are the execution points when a method is executed on that object. Therefore, this appearing in the advice body refers to that MyBusinessTask object although it refers to an aspect instance in AspectJ. If needed, the advice body can access private fields and methods in MyBusinessTask. This is not allowed in AspectJ unless the aspect is privileged. On the other hand, the advice body in GluonJ cannot access private fields or methods in Logger. The visibility scope is determined by the execution context of the advice body. In AspectJ, it is an instance of the aspect while it is the same context as the join point in GluonJ.

A unique feature of GluonJ is that an aspect implementation must be explicitly constructed in the aspect. In our example, a Logger object was constructed in the statement surrounded with the injection tag. Then it is associated with the MyBusinessTask object and used in the advice body. If program execution reaches a join point specified by a pointcut, the advice body is executed and it explicitly calls a method on the associated aspect implementation, that is, the Logger object. Note that GluonJ never instantiates an aspect since the aspect is glue code in GluonJ. From the implementation viewpoint, the code snippet in the aspect is merged into the methods of the aspect target, that is, MyBusinessTask.

3.2 Using the injection Tag for Dependency Injection

An advice body in GluonJ can be any Java code. It does not have to call aspectOf. For example, if a MyBusinessTask object had a field and that field refers to a Logger object, an advice body could call the log method on the object referred to by that field instead of the object returned by aspectOf.

The special form aspectOf and the injection tag are provided for adding a new field to an existing class while avoiding naming conflict. An aspect can give a specific name to an added new field, for example, by the following description:

```
<injection>
  Logger MyBusinessTask.link = new Logger();
</injection>
```

This adds a new field named link to the MyBusinessTask class and it initializes the value of that field so that it refers to a Logger object. The type of that field is Logger. However, this may cause naming conflict if another aspect adds a link field to the MyBusinessTask class.

If a special keyword aspect is specified as the name of the added field, this field becomes an anonymous field, that is, a field that has no name. An anonymous field can be accessed only through the special form aspectOf. For example, Logger.aspectOf(p) represents the anonymous field that is Logger type and belongs to the object p. We do not have to manually choose a unique field name for avoiding naming conflict.

There is also another rule with respect to the name of a newly added field. If the specified field name is the same as an already existing field in the same class, a new field is never added to the class. The initial value specified in the block surrounded with injection is assigned to that existing field with the same name.

This rule allows us to describe dependency injection with a simple aspect. For example, the example shown in the previous section can be described with the following aspect:

```
<aspect>
  <injection>
    Database MyBusinessTask.db = new MySQL();
  </injection>
</aspect>
```

This aspect specifies that a MySQL object is constructed and assigned to the db field in MyBusinessTask when an MyBusinessTask object is constructed. Since the db field already exists, no new field is added to MyBusinessTask. The aspect does not have to include a pointcut for picking out the construction of a MyBusinessTask object.

Although the block surrounded with the injection tag is similar to the intertype field declaration of AspectJ, it is not the same language element as the intertype field declaration. The added fields in GluonJ are private fields only accessible in the class to which those fields are added. On the other hand, private fields added by intertype field declarations of AspectJ are not accessible from the class to which those fields are added. They are only accessible from the aspect (implementation) that declares those fields.

3.3 Dependency Reduction

GluonJ was designed particularly for addressing inter-component dependency, which is a common goal to aspect-oriented programming and dependency injection. Thus GluonJ provides mechanisms for dealing with the two sources of the dependency: connections and method calls among components.

A component depends on another component if the former has a connection to the latter (i.e. the former has a reference to the latter) and/or the former calls a method on the latter. This dependency becomes a problem if the latter component implements a crosscutting concern. Let us call the former component *the caller* and the latter one *the callee*. In the example in Section 3.1, the caller is MyBusinessTask and the callee is Logger.

The inter-component dependency makes it difficult to reuse the caller-side component *as is*. If the callee is a crosscutting concern, it is not a sub-component of the caller; it is not contained in the caller or invisible from the outside of the caller. Therefore, those components should be independently reused without each other. For example, since Logger is a crosscutting concern and hence it is not crucial for implementing the function of MyBusinessTask, MyBusinessTask may be reused without Logger. Reusing the callee without the caller is easy; the program of that component can be reused *as is* for other software development. On the other hand, in regular Java, reusing the caller without the callee, for example, reusing MyBusinessTask without Logger needs to edit the program of the caller-side component MyBusinessTask since it includes method calls to the callee. These method calls must be eliminated from the program before the component is reused.

Connections Among Components: For reducing dependency due to connections among components, GluonJ provides the block surrounded with the injection tag. Although this dependency can be reduced with the technique of dependency injection, GluonJ enables framework independence discussed in Section 2.2 since it is an aspect-oriented programming (AOP) system. Furthermore, GluonJ provides direct support for expressing this dependency although in other AOP systems this dependency is indirectly expressed by advice intercepting object creation. We adopted this design of GluonJ because addressing the de-

pendency due to inter-component connections is significant in the application domain of GluonJ.

Method Calls Among Components: For reducing dependency due to method calls, GluonJ provides the pointcut-advice architecture. For example, as we showed in Section 3.1, the dependency between MyBusinessTask and Logger due to the calls to the log method can be separately described in the block surrounded with the advice tag. This separation makes the method calls *implicit* and *non-invasive* and thus MyBusinessTask will be reusable independently of Logger. The reuse does not need editing the program.

Note that the method call on the Database object within the body of MyBusinessTask in Section 2.1 does not have to be implicit by being separately described in XML. This call is a crucial part of the function of MyBusinessTask and hence MyBusinessTask will never be reused without a component implementing the Database interface. We do not have to reduce the dependency due to this method call.

Since the pointcut-advice architecture of GluonJ was designed for reducing dependency due to method calls, the aspect implementation that a method is called on is explicitly specified in the advice body written in Java. That aspect implementation can be any regular Java object. It can be an object constructed in the block surrounded with injection but, if needed, it can be any other object. It is not flexible design to enable calling a method only on the aspect implementation that the runtime system implicitly constructs and associates with the aspect target. We revisit this issue in Section 4.

3.4 The Tags of GluonJ

A block surrounded with the aspect tag may include blocks surrounded with either the injection tag or the advice tag. We below show brief overview of the specifications of these tags.

Injection Tag: In a block surrounded with the injection tag, an anonymous field can be declared. For example, the following declaration adds a new anonymous field to the MyBusinessTask class:

```
<injection>
  Logger MyBusinessTask.aspect = new Logger(this);
</injection>
```

The initial value of the field is computed and assigned right after an instance of MyBusinessTask is constructed. The expression computing the initial value can be any Java expression. For example, it can include the this variable, which refers to that MyBusinessTask object in the example above.

If the declaration above starts with static, then a static field is added to the class. The initial value is assigned when the other static fields are initialized.

The field added by the declaration above is accessible only in the aspect. To obtain the value of the field, the special form aspectOf must be called. For example, Logger.aspectOf(t) returns the Logger object stored in the anonymous

field of the MyBusinessTask object specified by t. If the anonymous field is static, then the parameter to aspectOf must be a class name such as MyBusinessTask.

A real name can be given to a field declared in an injection block. If an valid field name is specified instead of aspect, it is used as the name of the added field. That field can be accessed with that name as a regular field in Java. If there already exists the field with that specified name, a new field is not added but only the initial value specified in the injection block is assigned.

An anonymous field can be added to an object representing a control flow specified by the cflow pointcut designator. This mechanism is useful to obtain similar functionality to a percflow aspect instance in AspectJ. To declare such a field, the aspect should be something like this:

```
<injection>
  Logger Cflow(call(* MyBusinessTask.*(..)).aspect
      = new Logger();
</injection>
```

An anonymous field is added to an object specified by Cflow. It represents a control flow from the start to the end of the execution of a method in MyBusinessTask. It is automatically created while the program execution in that control flow. To obtain the value of this anonymous field, aspectOf must be called with the thisCflow special variable. For example,

```
Logger.aspectOf(thisCflow).log();
```

aspectOf returns the Logger object stored in thisCflow. thisCflow refers to the Cflow object representing the current control flow.

An anonymous field can be used to associate a group of objects with another object. This mechanism provides similar functionality to the association aspects [17]. For example,

```
<injection>
  Logger MyBusinessTask.aspect(Session) = new Logger(this, args);
</injection>
```

This declaration associates multiple Logger objects with one MyBusinessTask. this and args are special variables. These Logger objects are identified by a Session object given as a key. The type of the key is specified in the parentheses following aspect. Multiple keys can be specified. The associated objects are obtained by aspectOf. For example,

```
Logger.aspectOf(task, session).log();
```

This statement calls the log method on the Logger object associated with a combination of task and session. aspectOf takes two parameters: the first parameter is a MyBusinessTask object and the second one is a Session object. aspectOf returns an object associated with the combination of these objects passed as parameters. If any object has not been associated with the given combination, aspectOf constructs an object and associates it with that combination. In other words, an

associated object is never constructed until aspectOf is called. In the case of the example above, a Logger object is constructed with parameters this and args. this refers to the first parameter to AspectOf (*i.e.* the MyBusinessTask object) and args refers to an array of Object. The elements of this array are the parameters to aspectOf except the first one. In this example, args is an array containing only the Session object as an element.

Advice Tag: A block surrounded with the advice tag consists of a pointcut and an advice body. The pointcut is specified by the pointcut tag. The syntax of the pointcut language was borrowed from AspectJ although the current implementation of GluonJ does not support the if and adviceexecution pointcut designators. Although && and || must be escaped, AND and OR can be used as substitution. The current implementation of GluonJ has neither supported a named pointcut. A pointcut parameter is defined by using the param tag. For example, the following aspect uses an int parameter i as a pointcut parameter. It is available in the pointcut and the advice body.

```
<advice>
  <param><name>i</name><type>int</type></param>
  <pointcut>
    execution(* MyBusinessTask.*(..)) AND args(i)
  </pointcut>
  <after>
    Logger.aspectOf(this).log(i);
  </after>
</advice>
```

An advice body can be before, after, or around. It is executed before, after, or around the join point picked out by the pointcut. Any Java statement can be specified as the advice body although the < and > operators must be escaped since an advice body is written in an XML file. A few special forms aspectOf(), thisCflow, and thisJoinPoint are available in the advice body. The thisCflow variable refers to a Cflow object representing the current control flow. The thisJoinPoint variable refers to an object representing the join point picked out by the pointcut. If the proceed method is called on thisJoinPoint, it executes the original computation at the join point. The return type of proceed() is Object. The proceed method is only available with around advice.

Reflection: Although aspectOf is available only in a advice body, GluonJ provides a reflection mechanism [18] for accessing anonymous fields from regular Java objects. Table 1 lists the static methods declared in Aspect for reflective accesses.

4 Comparison to AspectJ

Although GluonJ has borrowed a number of ideas from AspectJ, there are a few significant differences between them. The first one is the visibility rule. The

Table 1. The static methods in the Aspect class

void add(Object target, Object aspect, Class clazz) assigns aspect to an anonymous field of target. clazz represents the type of the anonymous field.
void add(Object target, Collection aspects, Class clazz) associates all the elements in aspects with target. clazz represents the class of the associated elements.
Object get(Object target, Class clazz) obtains the value of an anonymous field of target. clazz represents the type of the anonymous field.
Collection getAll(Object target, Class clazz) obtains the collection associated with target. clazz represents the type of the collection elements.
void remove(Object target, Object aspect, Class clazz) unlinks aspect associated with target. clazz represents the type of the anonymous field.
void remove(Object target, Collection aspects, Class clazz) unlinks all the elements in aspects associated with target. clazz represents the type of the collection elements.

advice body in GluonJ can access private members of the aspect target since it is glue code. On the other hand, the advice body in AspectJ cannot access except the members added by the intertype declarations. This is because the advice body in AspectJ belongs to the aspect implementation.

Another difference is how to specify which aspect implementation is associated with an aspect target. This section illustrates comparison between GluonJ and AspectJ with respect to this issue. Although GluonJ is similar to JBoss AOP and AspectWerkz rather than AspectJ, we compare GluonJ to AspectJ since the readers would be more familiar to AspectJ. In fact, AspectJ, JBoss AOP, and AspectWerkz are based on the same idea with respect to the association of aspect implementations. Note that, like GluonJ, JBoss AOP and AspectWerkz separate aspect bindings in XML from aspect implementation in Java. Although their aspect implementations are Java objects, they are implicitly constructed and associated as in AspectJ. On the other hand, an aspect implementation in GluonJ is explicitly constructed and associated.

4.1 Example

To illustrate that explicit association of aspect implementations in GluonJ enables a better expression of inter-component dependency than AspectJ, we present an implementation of simple caching mechanism in AspectJ and GluonJ. If a method always returns the same value when it is called with the same arguments, the returned value should be cached to improve the execution performance. Suppose that we would like to cache the result of the doExpensiveJob method in the following class:

```
public class MyTask {
  private int sessionId;
  public MyTask (int id) {
    sessionId = id;
  }
  public String doExpensiveJob(String s) {
    // the execution of this method takes a long time.
    // the result is computed from s and sessionId.
  }
}
```

Note that the returned value from doExpensiveJob depends only on the parameter s and the sessionId field. Thus we share cache memory among MyTask objects with the same session id.

We below see how GluonJ and AspectJ express the dependency between MyTask and the caching component. The goal is to implement the caching component to be independent of MyTask and naturally connect the two components by an aspect.

4.2 GluonJ

We first show the implementation in GluonJ (Figure 3). The following is the class for a caching component:

```
public class Cache {
  private HashMap cache = new HashMap();
  public Object getValue(JoinPoint thisJoinPoint, Object arg) {
    Object result = cache.get(arg);
    if (result == null) {
      try {
        result = thisJoinPoint.proceed();
        cache.put(arg, result);
      } catch (Throwable e) {}
    }
    return result;
  }

  // create a cache for each session.
  private static HashMap cacheMap = new HashMap();
  private static Cache factory(int sessionId) {
    Integer id = new Integer(sessionId);
    Cache c = (Cache)cacheMap.get(id);
    if (c == null) {
        c = new Cache();
        cacheMap.put(id, c);
    }
```

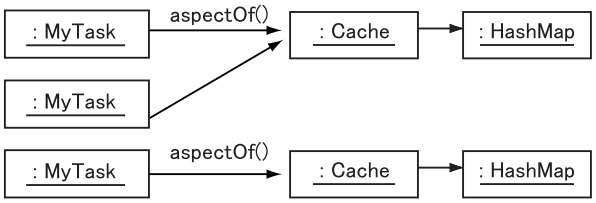

Fig. 3. The caching component in GluonJ

```
    return c;
  }
}
```

This component holds a hash table for caching the value returned from a method. factory is a factory method for constructing a Cache object for each session.

The Cache component is associated with a MyTask object. This association is described in the following aspect:

```
<aspect>
  <injection>
    Cache MyTask.aspect = Cache.factory(this.sessionId);
  </injection>
  <advice>
    <param><name>s</name> <type>String</type></param>
    <pointcut>
      execution(String MyTask.doExpensiveJob(..)) AND args(s)
    </pointcut>
    <around>
      return (String)Cache.aspectOf(this)
                          .getValue(thisJoinPoint, s);
    </around>
  </advice>
</aspect>
```

This aspect adds an anonymous field to MyTask. The value of this field is a Cache object for the session that the MyTask object belongs to. Then, if the doExpensiveJob method is executed, this aspect calls the getValue method on the associated Cache object.

Note that a Cache object is explicitly constructed in the aspect by calling a factory method. It is thereby associated with multiple MyTask objects belonging to the same session. The resulting object graph in Figure 3 naturally represents that the caching concern is per-session cache.

4.3 AspectJ (Using Intertype Declaration)

The caching mechanism can be also implemented in AspectJ. However, since AspectJ does not allow us to associate an aspect instance with a group of MyTask objects belonging to the same session, we must implement the per-session cache with a little bit complex programming. This is an example of the inflexibility for the implicit association of aspect instances in AspectJ. The following is an implementation using a singleton aspect and intertype field declaration (Figure 4):

```
privileged aspect CacheAspect {
  private HashMap MyTask.cache;    // intertype declaration

  after(MyTask t): execution(MyTask.new(..)) && this(t) {
    t.cache = factory(t.sessionId);
  }

  String around(MyTask t, String s): this(t) && args(s)
                && execution(String MyTask.doExpensiveJob(..)) {
```

```
    String result = (String)t.cache.get(s);
    if (result == null) {
      result = proceed(t, s);
      t.cache.put(s, result);
    }
    return result;
  }

  // create a cache for each session.
  private static HashMap cacheMap = new HashMap();
  private static HashMap factory(int sessionId) {
    Integer id = new Integer(sessionId);
    HashMap map = (HashMap)cacheMap.get(id);
    if (map == null) {
      map = new HashMap();
      cacheMap.put(id, map);
    }
    return map;
  }
}
```

Although the CacheAspect looks similar to the implementation in GluonJ, the resulting object-graph is different. It is far from the natural design. A single caching component, which is an instance of CacheAspect, manages the hash tables for all the sessions while each caching component in GluonJ manages a hash table for one session. Since there is only one caching component in AspectJ, a hash table for each MyTask object is stored in the cache field of the MyTask object. cache is the field added by intertype declaration. Hence the implementation of the caching concern is not only encapsulated within CacheAspect but also cutting across MyTask. Since AspectJ is a powerful aspect-oriented language, the implementation is not cutting across multiple components at the source-code level; it is cleanly modularized into CacheAspect. However, at the design level, the implementation of the caching concern involves MyTask. The developer must be aware that a hash table is contained in not CacheAspect but MyTask.

Another problem is that the caching concern is not really separated from other components since the dependency description (*i.e.* pointcut and advice) is

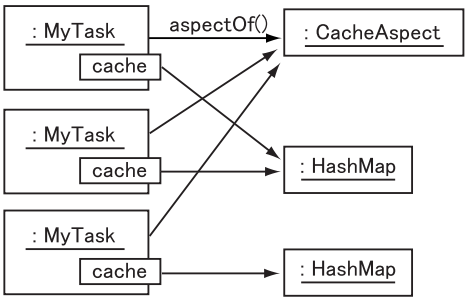

Fig. 4. The caching aspect using intertype declaration

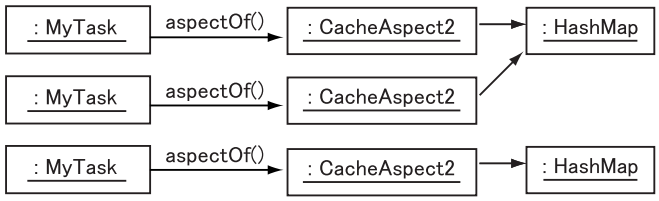

Fig. 5. The caching aspect using perthis

contained in the caching component. The caching component depends on MyTask since the class name MyTask is embedded in the intertype declaration in CacheAspect. If we reuse CacheAspect with another class other than MyTask, we must modify the definition of CacheAspect so that the cache field is added to that class. Although AspectJ provides abstract pointcut for parameterizing a class name occurring in a pointcut definition, it does not provide such a parameterization mechanism for intertype declarations.

Finally, since this aspect must access the sessionId field, which is private, it is declared as being privileged. A privileged aspect is not subject to the access control mechanism of Java. Thus, this implementation violates the encapsulation principle.

4.4 AspectJ (Using perthis)

The caching concern can be implemented with a perthis aspect (Figure 5). In the following implementation, an instance of CacheAspect2 is constructed for each MyTask object. This policy of aspect instantiation is specified by the perthis modifier. See the following program:

```
privileged aspect CacheAspect2 perthis(execution(* MyTask.*(..)) {
  private HashMap cache;    // aspect member

  after(MyTask t) : execution(MyTask.new(..)) && this(t) {
    cache = factory(t.sessionId);
  }

  String around(String s)
    : execution(String MyTask.doExpensiveJob(..)) && args(s) {
    String result = (String)cache.get(s);
    if (result == null) {
      result = proceed(s);
      cache.put(s, result);
    }
    return result;
  }

  // create a cache for each session.
  //     :
  // (the same as the factory method in CacheAspect)
}
```

Note that the hash table is stored in the cache field of the aspect instance. This aspect does not include intertype declaration. The cache field is a member of this aspect itself.

This implementation is simpler than the previous one since an instance of CacheAspect2 manages only one hash table stored in a field of that instance. CacheAspect2 does not have to access a field in MyTask. However, this implementation produces redundant aspect instances. The role of each aspect instance is merely a simple bridge between a MyTask object and a hash table. It has nothing elaborate. This is not appropriate from the viewpoint of either program design or efficiency.

Note that, in this implementation, both the caching component and the dependency description (with pointcuts and advice) are also tangled in CacheAspect2. However, separating the dependency description from the program of the caching component is not difficult if abstract pointcuts are used. We can define an aspect only for the caching mechanism and then define another aspect that extends the former aspect and implements the abstract pointcut for describing the dependency. The perthis modifier must be defined in the latter aspect.

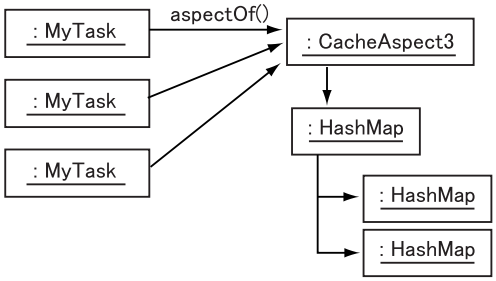

Fig. 6. The caching aspect using a hash table

4.5 AspectJ (Using a Hash Table)

The implementation we finally show uses a singleton aspect but it does not use an intertype field declaration or an aspect member field. In this implementation, either MyTask or CacheAspect3 do not include the cache field. The hash table is obtained from the factory method when the around advice is executed (Figure 6):

```
privileged aspect CacheAspect3 {
  String around(MyTask t, String s): this(t) && args(s)
                && execution(String MyTask.doExpensiveJob(..)) {
    HashMap cache = factory(t.sessionId);  // obtain from factory()
    String result = (String)cache.get(s);
    if (result == null) {
      result = proceed(t, s);
      cache.put(s, result);
    }
    return result;
  }
```

```
  // create a cache for each session.
  //     :
  // (the same as the factory method in CacheAspect)
}
```

This would be the best implementation among the three AspectJ-based ones. The caching aspect is separated and independent of MyTask. No redundant aspect instance is produced. However, it is never highly efficient to call the factory method whenever the doExpensiveJob method is executed. Furthermore, this centralized design of caching mechanism is implementation-oriented. It would not be the design easily derived after the modeling phase. The easily derived design would be something like Figure 3 achieved by GluonJ. Figure 6 shown here would be the design that we could obtain by modifying that easily derived design to be suitable for implementation in a particular language.

Note that, in the implementation shown above, the dependency description (with pointcuts and advice) is also tangled with the caching component. However, separating the dependency description from the program of the caching component is possible by using abstract pointcuts.

5 Related Work

There are a number of aspect-oriented languages and frameworks that separate aspect binding and aspect implementation. Like GluonJ, JBoss AOP [9] and AspectWerkz [1] uses XML for describing aspect binding while Aspectual Components [13], Caesar [14] and JAsCo [19] uses extended language constructs. JAC [16] uses a programming framework in regular Java. Even AspectJ provides abstract aspects for this separation [8]. However, these systems allow only implicit association of an aspect implementation and hence they have a problem discussed in this paper. An aspect implementation is automatically constructed and implicitly associated with the aspect target in the specified scheme such as issingleton and perthis of AspectJ. Although JBoss AOP provides customization interface in Java for extending the behavior of perthis, it complicates the programming model.

The dynamic weaving mechanism of Caesar [14] allows associating an aspect implementation at runtime when the developers specify. It provides better flexibility but an aspect implementation is still automatically constructed and implicitly associated with the aspect target.

Association aspect [17] allows implementing a crosscutting concern by an explicitly constructed instance of an aspect. It is an extension to AspectJ and it is a language construct focusing on associating an aspect instance to a tuple of objects. GluonJ can be regarded as a framework generalizing the idea of association aspect and applying it to dependency reduction among components.

The implicit association of an aspect implementation (and an aspect instance in AspectJ) might be the ghost of the metaobject protocol [11], which is one of the origins of aspect-orientated programming. Although this design is not a problem if an aspect crosscuts only a single other concern, it should be revised

to fully bring out the power of aspect orientation. Otherwise, advantages of aspect-oriented programming might be small against metaobject protocols.

AspectJ2EE [4] is an aspect-oriented programming system for J2EE. It restricts an aspect implementation to being associated with only a single aspect target. Therefore, it has the problem discussed in this paper.

Alice [5], JBoss AOP [9], and AspectWerkz [1] allow pointcuts that capture Java 5 annotations. This feature can be used for performing dependency injection on the fields annotated with @inject. Although this provides better syntax support, the developers must still define an aspect like DependencyInjection shown in Section 2.2.

The branch mechanism of Fred [15] provides basic functionality of aspect-oriented programming. It is similar to GluonJ since both of them provide only a dispatching mechanism based on pointcut and advice but they do not allow instantiation of aspects unlike AspectJ. However, Fred is a very simple Scheme-based language and it provides only a limited mechanism for dealing with dependency among components.

6 Conclusion

Reducing inter-component dependency is the goal of dependency injection but aspect-oriented programming can give a better solution to this goal. However, existing aspect-oriented programming systems have a problem. They can express only limited kinds of dependency relation since they implicitly associate an aspect implementation with an aspect target. The developers cannot fully control this relation.

To address this problem, this paper proposed *GluonJ*, which is our aspect-oriented framework for Java. A unique feature of GluonJ is that an aspect implementation is explicitly associated with aspect targets. An aspect in GluonJ consists of pointcuts and glue code written in Java. This glue code explicitly constructs an aspect implementation and associates it with appropriate aspect targets. The aspect implementation in GluonJ is a regular Java object.

We have implemented a prototype of GluonJ as a bytecode translator built on top of Javassist [2]. It supports most pointcut designators of AspectJ except cflow, which will be implemented in near future.

References

1. Boner, J., Vasseur, A.: AspectWerkz 1.0. `http://aspectwerkz.codehaus.org/` (2002)
2. Chiba, S.: Load-time structural reflection in Java. In: ECOOP 2000. LNCS 1850, Springer-Verlag (2000) 313–336
3. Chiba, S., Masuda, T.: Designing an extensible distributed language with a meta-level architecture. In: Proc. of the 7th European Conference on Object-Oriented Programming. LNCS 707, Springer-Verlag (1993) 482–501

4. Cohen, T., Gil, J.Y.: AspectJ2EE = AOP + J2EE : Towards an aspect based, programmable and extensible middleware framework. In: Proceedings of the European Conference on Object-Oriented Programming. Number 3086 in LNCS (2004) 219–243
5. Eichberg, M., Mezini, M.: Alice: Modularization of middleware using aspect-oriented programming. In: Software Engineering and Middleware (SEM) 2004. (2004)
6. Fowler, M.: Inversion of control containers and the dependency injection pattern. `http://www.martinfowler.com/articles/injection.html` (2004)
7. Golm, M., Kleinöder, J.: Jumping to the meta level, behavioral reflection can be fast and flexible. In: Proc. of Reflection '99. LNCS 1616, Springer (1999) 22–39
8. Hannemann, J., Kiczales, G.: Design pattern implementation in java and aspectj. In: Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications. (2002) 161–173
9. JBoss Inc.: JBoss AOP 1.0.0 final. `http://www.jboss.org/` (2004)
10. Johnson, R., Hoeller, J.: Expert One-on-One J2EE Development without EJB. Wrox (2004)
11. Kiczales, G., des Rivières, J., Bobrow, D.G.: The Art of the Metaobject Protocol. The MIT Press (1991)
12. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: ECOOP 2001 – Object-Oriented Programming. LNCS 2072, Springer (2001) 327–353
13. Lieberherr, K., Lorenz, D., Mezini, M.: Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA (1999)
14. Mezini, M., Ostermann, K.: Conquering aspects with caesar. In: Proc. of Int'l Conf. on Aspect-Oriented Software Development (AOSD'03), ACM Press (2003) 90–99
15. Orleans, D.: Incremental programming with extensible decisions. In: AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development, ACM Press (2002) 56–64
16. Pawlak, R., Seinturier, L., Duchien, L., Florin, G.: Jac: A flexible solution for aspect-oriented programming in java. In: Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001). LNCS 2192, Springer (2001) 1–24
17. Sakurai, K., Masuhara, H., Ubayashi, N., Matsuura, S., Kimoya, S.: Association aspects. In: Aspect-Oriented Software Development. (2004) 16–25
18. Smith, B.C.: Reflection and semantics in Lisp. In: Proc. of ACM Symp. on Principles of Programming Languages. (1984) 23–35
19. Suvée, D., Vanderperren, W., Jonckers, V.: Jasco: An aspect-oriented approach tailored for component based software development. In: Proc. of Int'l Conf. on Aspect-Oriented Software Development (AOSD'03), ACM Press (2003) 21–29
20. Welch, I., Stroud, R.: From dalang to kava — the evolution of a reflective java extension. In: Proc. of Reflection '99. LNCS 1616, Springer (1999) 2–21