

Typecasting As a New Join Point in AspectJ

M. Devi Prasad

Manipal Center for Information Science, Manipal Academy of Higher Education

Manipal -576119, Karnataka, India

devi.prasad@mahe.manipal.edu

Telephone: +91-08252-573491

ABSTRACT

AspectJ does not treat typecasting as an important operation in program's execution. However, down casting and cross casting play an important role in overall behavior of a java program. We demonstrate that Java programs use type casting primarily for retrieving references to new types from distantly related ones. We investigate AspectJ's inadequacy in selecting execution points that use typecasting to yield new object or interface references. We demonstrate the necessity for a "reference creation by type casting" joinpoint and argue that its addition makes AspectJ's existing model more expressive. We bring out characteristics of such a joinpoint and illustrate its usage.

1. INTRODUCTION

AspectJ [1] is a general purpose programming language based on Java for modularizing crosscutting concerns [3]. Often, crosscutting concerns are features of an application, including (but not limited to) logging, tracing, synchronization, and caching. Experience shows that implementing these features in traditional OO languages results in scattered and/or tangled code [3]. AspectJ introduces new modularizing construct named as 'aspect'. An aspect helps in expressing new behavior in an additive manner over and above the existing OO implementation. It does so by providing language level expressions to identify and augment key structural and behavioral elements in the underlying OO system.

Essentially, aspects are useful in providing incremental extensions to components. Since components are accessed using well known interfaces, any conceived functionality enhancement achieved by aspects has to be interface centered and must be transparent to existing client code. Additionally, the new functionality may necessitate sharing state and information among members of the introduced abstraction. Some design patterns, such as Decorator or Proxy [4], are helpful in modularizing certain concerns. AspectJ can be effectively used to create design patterns [5]. When AspectJ is employed for this purpose, some important consequences of its join point model should be borne in mind:

1. When all methods of a class or interface require distinct advice, declaring pointcuts designating individual methods and providing separate advice for each pointcut is both unwieldy and undesirable. They tend to be fragmented.

2. Only around advice can define a return type for advice body. Therefore, around advice can be employed to wrap a newly retrieved interface or object reference. It can then return the reference to this wrapper instead of the wrapped reference.

Statement (2) above suggests that by suitably advising around the join points that create a new instance or obtain an interface reference, we can avoid fragmented advice constructs. Using decorators gives an OO flavor compared to separate around advice modularized in an aspect. The former lends itself to the benefits of behavioral extension by inheritance.

Such a solution relies on the expressiveness of the join point model to pick out object or interface reference creation points. In Java programming language, three kinds of expressions can generate a new reference: (1) the ‘new’ operator call, (2) (factory) method call, and (3) type casting expression. AspectJ can directly express only the first two join points among the three listed here.

In this paper we show that reference generation by type casting is an important programming technique used in practical programs. We argue that using AspectJ, creating decorator like wrappers is not possible in cases that use down casting or cross casting expressions. We propose a new join point to specifically capture type casting expressions that generate new references to objects or interfaces.

The rest of the paper is organized as follows: In section 2 we give a motivating example to illustrate the problem. In section 3 we show inadequacy of AspectJ in providing a simple and effective solution to this problem. In section 4 we propose a new joinpoint for capturing reference creation with type casting that improves AspectJ’s expressiveness. In section 5 we discuss a prototypical implementation of this idea. We conclude section 6 by summarizing the results and discuss intended future work.

2. ILLUSTRATIVE EXAMPLE

In this section we introduce an example for reference in the subsequent sections. In this example we consider a distributed service implemented using the Java RMI infrastructure. The remote server component implements two interfaces, **InStream** and **OutStream**. The declaration of interfaces and a typical client program is also shown here. Since the remote component implementation is irrelevant to this discussion, it is not produced here. Details such as error handling or remote exception processing are not shown for brevity.

```
interface InStream extends Remote {  
    int  available();  
    void close();  
    int  read();  
    int  read(byte[] data);  
}
```

```
interface OutStream extends Remote {  
    void flush();  
    void close();  
    int  write(int data);  
    int  write(byte[] data);  
}
```

<pre> class Consumer { public long streamDataIn(Remote r) { long totalStreamed = 0; // get required interface reference InStream in = (InStream) r; // use 'in' to stream data & update // 'totalStreamed' ... return totalStreamed; } public void streamDataOut(Remote r) { // get required interface reference OutStream out = (OutStream) r; // use 'out' to stream data ... } } </pre>	<pre> // client of the Java RMI component... import java.rmi.*; class main { public static void main(String args[]) { Remote r = Naming.lookup ("rmi://server/ServiceProviderImpl"); Consumer c = new Consumer(); c.streamDataIn(r); c.streamDataOut(r); ... } } </pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

A client streams data from the remote service using the **InStream** interface and it sends out data using the **OutStream** interface. The **Consumer** class implements the interaction between the client and the service. In Java RMI, each remote interface must inherit from a tagging interface named **Remote**. Its sole purpose is to advertise to the RMI runtime that the interface can be used for remote invocations.

Given such a component with well-known interfaces, we are interested in providing, say, a feature like client side caching for improved performance. Moreover, we expect this feature to be supplemented transparently to the client code shown here. Our aim will be to determine the limitations in AspectJ's expressiveness in handling issues that are unique to implementations represented by this example. We will also be interested in improving AspectJ's vocabulary to deal with these discrepancies.

3. INADEQUACY OF AspectJ JOIN POINT MODEL

Consider the case where the client wishes to provide caching responsibilities to **OutStream** object. For this, the caching strategy must coordinate buffer accesses among **OutStream** methods on the client side. For instance, many write operations on **OutStream** may update the local cache before requesting a **close** on the stream. However, new responsibilities of **close** include flushing the cache contents before actually closing the stream. Similarly when the local cache is full, invoking a write operation must ensure that the buffer is synced to the server before further caching.

3.1 Modeling caching concern using around advice constructs

As a first approximation, we try to encapsulate the caching concern as an aspect with unique around advice matching each method of the **OutputStream** interface.

```
aspect CacheOutputStream {
    byte []cache = new byte[...];
    int curOffset = 0;

    int around(OutputStream out, byte[] data):
        (target(out) && call(int write(..)) && args(data) && !within(CacheOutputStream) ) {
        if (cache does not have room for new data) {
            out.write(cache, curOffset);
            curOffset = 0;
        }
        // In any case, append 'data' to cache and update 'curOffset'
        copyAndUpdateOffset(data);
        // return # bytes actually written
        return ...
    }
    int around(OutputStream out, byte data):
        (target(out) && call(int write(..)) && args(data) && !within (CacheOutputStream)) {
        //similar to the version given above
        ...
    }
    void around(OutputStream out):
        (target(out) && call(void close()) && !within (CacheOutputStream)) {
        out.flush(); //if there is data in cache – need to flush it
        out.close();
    }
    void around(OutputStream out):
        (target(out) && call(void flush()) && !within (CacheOutputStream)) {
        if (curOffset > 0) { //if there is data in cache – need to flush it
            out.write(cache);
            out.flush();
        }
    }
}
```

It is evident from this example that specifying proper pointcuts poses a non trivial challenge. We should also be careful to avoid recursion in advice execution. In addition, we should take special care in declaring the pointcut to specifically avoid matching **OutputStream.write()** calls made within the aspect declaration.

We can also infer from this example that when it is necessary to impart additional behavior for an underlying module, all or a large number of elements belonging to that module might require enhancement. When methods of an interface demand coordinated behavior, advising individual methods with unique responsibilities becomes tedious. Even when such advices are provided, program readability or comprehension may get affected because it requires some effort for one to associate an advice with a method.

A decorator design pattern [4] represents an alternate solution in such contexts. A decorator provides additional behavior around an existing implementation by implementing the same interface(s) as the target object. So it conforms to the interface layout expected by the client.

3.2 Modeling caching concern using decorator and around advice constructs

In most of the cases, decorator can be created by advising around join points that generate objects or interface references. For example, the following around advice intercepts requests to create a new instance of RMI component and decorates the fresh object with a wrapper **RemoteServiceDecorator**. This wrapper object maintains reference to the original remote object. The decorator decides when certain calls need delegation and appropriately routes call to the remote object.

```
aspect CachingOutputStreamDecorator {  
    ...  
    Remote around() : call(public static Remote java.rmi.Naming.lookup(..)) {  
        Remote r = proceed();  
        Remote remoteDecorator = new RemoteServiceDecorator (r);  
        return remoteDecorator;  
    }  
}
```

We can set up similar advice around object creation joinpoints that use the ‘new’ operator. These two joinpoints (a factory method call and constructor call) represent important junctures in program execution where new object or interface references are either generated or transferred. With this arrangement, the aspect **CachingOutputStreamDecorator** trivially captures decorator creation concern. Actual caching concern is implemented by the **RemoteServiceDecorator**.

However, a careful analysis of this solution uncovers a serious problem. In the current example, the client can freely downcast or crosscast one reference to the other among **InStream**, **OutputStream** or **Remote** interface types. Since the around advice wraps remote object reference with a decorator object, the client obtains a **RemoteServiceDecorator** reference instead of remote object reference. This implies that this decorator should implement all interfaces that the remote component exposes. Otherwise, subsequent type cast to some expected remote interface in the client code would throw runtime exception. In our example, **RemoteServiceDecorator**, therefore, should implement a trivial version of **InStream** that simply forwards calls to remote object along with an implementation of **OutputStream** that actually encapsulates the caching concern.

Expecting clients to provide trivial implementation for interfaces of no interest to the application is not only counter intuitive but also inefficient. Given the simplicity of the problem at hand, we should be able to provide a simple and efficient solution.

3.3 Type cast as a major join point

In our RMI example, the **streamDataIn** and **streamDataOut** methods (of the **Consumer** class) are designed to work with one interface type, **InStream** and **OutStream** respectively. We wish to decorate only the **OutStream** interface, perhaps only inside **streamDataOut** method. However, **streamDataIn** and **streamDataOut** methods obtain reference to a sub type (**InStream** or **OutStream**) by down casting a reference variable of the super type (**Remote**). Because the remote object in our example implements both **InStream** and **OutStream**, a cross cast from **InStream** to **OutStream** is a perfectly valid operation.

Type casts of these two kinds are generally employed in traversing down and across type hierarchies. It is a common technique used in languages such as Java that lack support for templates or generic types. Even languages that support template types provide for down cast or cross cast. C++, for instance, provides three different flavors of type cast expressions. We can view these expressions as events that generate new information from the currently available one. Therefore they convey important purpose in program's execution.

AspectJ however does not treat down casting and cross casting to be interesting join points in a program's behavior. Thus, in AspectJ, there is no point cut expression that can pick out these two constructs for further advice. This also means that there is no way in AspectJ to create efficient and minimal decorator per interface if the program chooses "reference creation by down casting" pattern as illustrated in our example.

4. A PROPOSAL FOR NEW JOINPOINT

In this section, we give some salient features of a "reference generation with down (or cross) cast" join point and show its typical usage within AspectJ. We term this join point as a **reftrans** join point. It is a point in the program execution where

- A super type reference is down cast to a sub type
- Some interface or object reference is cross cast to another type

The above definition considers only reference variables as the source (i.e., right hand side (RHS)) of type casting. The two other kinds of expressions that can yield an interface or object reference are a method call or new operator. Since AspectJ supports them directly, there is no necessity to treat them special.

Here is a pointcut declaration that can pick out **reftrans** joinpoints. We assume that the **OutStreamDecorator** class provides caching support for the **OutStream** clients.

```
aspect RefTransDecorator {
    pointcut OutStreamFromRemote(Remote r) : args(r) && reftrans((OutStream) r);

    OutStream around(Remote remObj) : OutStreamFromRemote(remObj) {
        OutStreamDecorator outDecorator = new OutStreamDecorator((OutStream) remObj);
        return outDecorator;
    }
}
```

The emphasized part of pointcut declaration represents support for picking out a type cast from super type (**Remote**) to sub type (**OutputStream**). The declaration contains two parts: an **args** declarator from the AspectJ vocabulary combined with **reftrans**. The **args** declaration indicates source type of the cast expression. The **reftrans** pointcut specifies the argument picked out by **args** and the target type of cast expression. In this manner, the complete context for type casting is inferred. When a join point matches this declaration, AspectJ picks up the actual argument at that join point and makes it available at the advice. When the **RefTransDecorator** aspect (listed above) is applied to our example code, the statement

```
OutputStream out = (OutputStream) r;
```

in **streamDataOut** method of **Consumer** class matches the **OutputStreamFromRemote** point cut. The **remObj** parameter of the around advice is bound to reference **r** from the type casting statement. The advice body passes this reference to the **OutputStreamDecorator** constructor.

This simple example illustrates combining the **reftrans** join point with an around advice to create a wrapper for spontaneously created interface references. It provides a simple yet flexible alternate to writing many **after** and **before** advices for individual members of an interface. It gives developers more expressive power at a little added cost.

5. A PROTOTYPE IMPLEMENTATION WITHIN AspectJ

We have modified the AspectJ compiler [2] to support the proposed **reftrans** joinpoint. The support for this feature meshes well with existing syntax and semantics of AspectJ. A primitive pointcut representing **reftrans** is the only addition to the existing repertoire. Normal AspectJ point cut declaration (PCD) can combine this pointcut with other pointcuts in the usual manner.

6. CONCLUSION AND FUTURE DIRECTIONS

There are two major contributions in this paper. First, we have illustrated that in some cases generating a decorator for an object gives more flexibility than developing separate **before** and **after** advice for individual methods of an object. Second, we demonstrated the necessity to capture particular kinds of type cast expressions as important points in program execution. We showed how, under certain circumstances, the absence of a **reftrans** like join point in AspectJ causes inconvenience in creating decorator objects. In addition to these, we have provided a proof of concept implementation of this facility within AspectJ.

The current implementation does not handle certain special cases. For instance, in the following Java statement

```
InStream in = (InStream) (OutputStream) r;
```

type coercion is repeatedly applied to obtain same result as in the following pair of statements:

```
OutputStream out = (OutputStream) r; InStream in = (InStream) out;
```

The current implementation considers only those statements that down cast or cross cast a reference variable. I plan to study cases where repeated type coercion is meaningful and extend the implementation accordingly.

Java currently does not support operator overloading. In contrast, C++ allows operator overloading. In C++, one can overload various type coercion operators for a user-defined type. The translator for C++ wires up a call to the appropriate overloaded coercion method. I plan to study the implication operator overloading on the design of join points in general and to **reftrans** in specific. In addition, the proposed generics support for Java [6] will undoubtedly reduce the necessity of type coercion. On the other hand, for dynamic type discovery (as shown in the RMI example in this paper), type casting will continue to be an important language element. I plan to study the effect of generics on the join point model of AspectJ.

Elsewhere researchers have discussed the necessity of more expressive and precise join points for current generation of AOP languages [7]. This paper has suggested a “low-level” join point that works only at the code level. AspectJ is also a language with low-level support for specifying and composing crosscutting code into a core system. There is certainly a need for a means to separate crosscutting concerns across the lifecycle [8]. We have been working on a project that attempts to reverse engineer an AspectJ based software system into an extended UML model and a collection of OCL constraints [9].

7. ACKNOWLEDGEMENTS

The anonymous reviewers of the initially submitted version of this paper gave good advice on the content, organization and flow of material. Professor B.D. Chaudhari helped shaping the ideas in the initial stages and provided comments on the earlier drafts of this paper.

8. REFERENCES

1. AspectJ Programming Guide. <http://www.eclipse.org/aspectj/>
2. AspectJ 1.0.6 source code <http://www.eclipse.org/aspectj/>
3. Gregor Kiczales, et al., *Aspect-Oriented Programming*. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), June 1997.
4. E. Gamma, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Reading, MA: 1995
5. Jan Hannemann and Gregor Kiczales. *Design Pattern Implementation in Java and AspectJ*. OOPSLA - 2002.
6. JSR-000014, *Adding Generics to the JavaTM Programming Language*. <http://www.jcp.org/aboutJava/communityprocess/review/jsr014/>
7. Gregor Kiczales. *The Fun Has Just Begun*. Keynote Address, AOSD - 2003. (<http://aosd.net/archive/2003/kiczales-aosd-2003.ppt>)
8. Siobhan Clarke and Robert J. Walker. *Towards a Standard Design Language for AOSD*, AOSD - 2002.
9. Kleppe and J. Warmer. *The Object Constraint language, Precise modeling with UML*. Addison Wesley Professional.