Independent Extensibility for Aspect-Oriented Systems

Pascal Costanza, Günter Kniesel, and Michael Austermann
University of Bonn, Institute of Computer Science III
Römerstr. 164, D-53117 Bonn, Germany
{costanza|gk|austerm}@cs.uni-bonn.de

April 15, 2001

Abstract

So far, there is no satisfactory means to safely combine aspects that have been developed independently. In the general case, the process of jointly applying different aspects is not guaranteed to produce complete and unambiguous results. This paper shows that in the relevant case of pure interface modifications, completeness, uniqueness and independent extensibility can be ensured. This results in a strong reduction of the conceptual complexity of aspect combination. This idea has been incorporated into JMangler, a framework for load-time transformation of Java classes that can be used as a "back end" for arbitrary AOP systems.

1 Motivation

1.1 Unanticipated Adaptations

Software requirements are in a constant flux. Some changes in requirements can be anticipated by software developers, so that the necessary adaptations can be prepared for, for example by suitable parameterization or by application of dedicated design patterns. Within the scope of these anticipated options for adaptation, software can be used already in a flexible way, albeit in restricted limits and with a corresponding increase in development effort.

However, unanticipated changes of requirements occur repeatedly in practice, and the above suggested techniques cannot tackle them by definition. Furthermore, the manual integration of hooks for any conceivable eventuality is not a feasible option, since this dramatically decreases reliability, efficiency and maintainability.

Alternatively, programming languages and runtime systems should be equipped with features that allow for far reaching manipulations of program internals without destructively modifying its source code. This leads to an increase of options for unanticipated adaptation as well as a decrease of effort to prepare for anticipated adaptations. Therefore the structure of programs can be kept much simpler from the outset and they can be focussed on solving their primary tasks.

In order to provide for this significant simplification of software development, the goal of the TAILOR Project at the Institute of Computer Science III of the University of Bonn [10] is to conceive and implement enhancements of programming languages and runtime systems to allow for unanticipated adaptability of software. In doing so, special attention is payed to the issues of component-oriented software, among others.

1.2 Focus on Component-Oriented Software

The following definition for software components is quoted in [9]:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

This definition pushes the limits even further, both on the level of the components that are the targets of adaptation as well as on the level of the adaptations that are applied to the components.

- *Components* are usually deployed using a compiled format, and their source code is not available for modifications. Even if the source code can be accessed in some cases, direct modifications are still not feasible, since they would lapse in the presence of new versions of a component. However, it is still highly desirable to be able to use components in unanticipated contexts and with unanticipated adaptations.

- *Adaptations* should be expressible in a form that takes the shape of components, too. Here, the aim is to provide a mechanism that allows one to independently deploy dedicated adaptations and make them composable by third parties.

1.3 Challenges

A viable alternative to direct modifications of source code is the declaration of incremental program transformations that can be carried out automatically. In this context, there are two approaches which we regard as particularly important:

- **Aspect-Oriented Programming (AOP)** [1] enables one to group semantically related transformations into so-called *aspects*. These aspects are orthogonal to the traditional modularisation into classes and inheritance hierarchies.

- **Loadtime Component Adaptation** The seminal work on BCA [7] has shown that it is best to apply unanticipated adaptations when a program is being loaded, just before execution. In this way, classes can be transformed that are determined to be part of an application as late as runtime.

In the following sections we discuss weaknesses of existing AOP technology with regard to independent extensibility, a key feature of component-oriented software [9], and suggest a partial solution. We present a framework for loadtime transformations of Java programs that implements this solution. This framework can be used as a "back end" for arbitrary AOP systems.

2 Problems of Unanticipated Aspect Composition

So far, there is no satisfactory means to safely combine aspects that have been developed independently. In the general case, when such aspects have not been spefically designed for joint use and do not have intimate knowledge of their respective implementation details, unwanted side effects are likely to occur. Explicit composition of aspects without unwanted side effects is possible only if the joint use of these aspects has been anticipated. The challenge is to find an automatically applicable way of combining aspects without knowledge of their intimate details that still avoids unwanted side effects.

This problem can be divided into two parts that are discussed in the following subsections. In order not to tie this discussion to a particular AOP technique, we use the general term *program transformation*. To be more specific, we deal with *positively triggered program transformations*, which can be initiated only by the existence of a particular property of a program, but not by the absence of such a property. All AOP systems known to the authors are restricted to this kind of transformations.

2.1 Mutual Triggers

The first problem of unanticipated composition of aspects is the possible occurence of mutual triggers. A late transformation possibly adds properties to a program that requires the reapplication of transformations that have been executed already.

Mutual triggers cannot be dealt with in current AOP systems, since each transformation can be applied only once. This may result in potentially incomplete programs whose missing parts could be supplied only by reapplication of "old" transformations.

It seems that transformations should be iterated until a program is not changed anymore, that is until a fixed point is reached. In some cases this may be required even when a single transformation is to be employed, but the target program has a cyclic structure with regard to the properties that trigger the transformation's application. An example is given in section 5.

However, the need to iterate transformations immediately gives rise to the following questions.

- Does the iteration terminate?

- Is the result unambiguous?

2.2 Order of Transformations

Unfortunately, the answer to both questions is negative in the general case. The following example illustrates the different outcomes of the same transformations, depending on the order of their application.

Suppose that the following program is to be transformed by two aspects Access and Counter.

```
public class C {
   public B b = new B();

   public void manipulateB() {
      b.doSomething();
   }
}
```

The purpose of Access is to extend a class by access methods for each of its public fields, and to replace all direct accesses to these fields by calls to these generated methods. The purpose of Counter is to amend each field of a class by a counter that is incremented on each read access to its associated field.

As described above, these two aspects are applied alternately until class C is not changed anymore. Depending on which aspect comes first, this process leads to different outcomes, as shown in figure 1. The difference manifests itself in method manipulateB(). Its implementation in class C' increments the counter of field b, whereas this is not the case in class C". This difference is caused by the fact that in the second case, Access replaces the direct access to b by a call to the respective access method *before* Counter has a chance to recognize it. As a result, b_counter in C' is increased twice as often as in C".

```
    public class C' {
       public B b = new B();
(1)    private int b_counter = 0;

(2)    public void setB(B _b) {
          this.b = _b;
       }
(2)    public B    getB()    {
(3)       b_counter++;
          return this.b;
       }

       public void manipulateB() {
(1)       b_counter++;
(2)       getB().doSomething();
       }
    }
```

(1) Counter, (2) Access, (3) Counter, ...

```
    public class C'' {
       public B b = new B();
(2)    private int b_counter = 0;

(1)    public void setB(B _b) {
          this.b = _b;
       }
(1)    public B    getB()    {
(2)       b_counter++;
          return this.b;
       }

       public void manipulateB() {

(1)       getB().doSomething();
       }
    }
```

(1) Access, (2) Counter, ...

Figure 1: Aspects may have differing results, depending on the order of their application.

3 Partitioning of Transformations

Besides illustrating this negative result, this example is apt for another important observation. Whereas the resulting *method body* of manipulateB is dependent on the order of transformations, the *interfaces* of C' and C" are exactly the same, independent of this order.

In fact, it can be shown that the result of transformations of interfaces is always independent of the order of transformations. This is due to the fact that interfaces are *unordered sets* of names and signatures, whereas implementations are *ordered lists* of statements and expressions. It is obvious that the order in which elements are added to a set does not affect the resulting set, whereas the order in which a list is manipulated is essential for the result.

For this reason, program transformations should be partitioned into two classes, namely

- code transformations, and
- interface transformations.

Code transformations modify existing method bodies, *interface transformations* are responsible for everything else, for example adding classes, inheritance relations, fields and methods.

Contrary to fields, which are assigned default values by Java when no field initializer is given, methods cannot be given any meaningful "default behavior" in the general case. Therefore, when adding non-abstract methods during interface transformations, they must be supplied with initial code which can be transformed further by code transformations. Therefore, the addition of a method including an initial method body is still regarded as a pure interface transformation.

Likewise, changes to modifiers are regarded as pure interface transformations. However, it is ensured that accessibility (public, protected, etc.) of members is widened only.

Generally speaking, only *monotone* interface transformations are considered valid – they are allowed to add new members only, or widen the accessibility of existing members, but they must not remove or change members, or narrow their accessibility. Modifications may be expressed as code transformations only.

With regard to our original problem statement these considerations can be summed up as follows:

- Aspects that are restricted to interface transformations can be developed independently and combined automatically.
- For the time being, aspects that include code transformations cannot be automatically combined.

Based on these observations, a tool for loadtime transformations has been developed in the course of the TAILOR project. We present this tool in the following section.

4 JMangler

JMangler is a framework that is implemented in Java and that allows transformations to be applied to Java classes during loadtime, just before these classes are linked by the Java Virtual Machine. Programmers can write their own *transformer components*[1] that analyze the classes

[1] These transformer components are equivalent to aspects in other AOP systems.

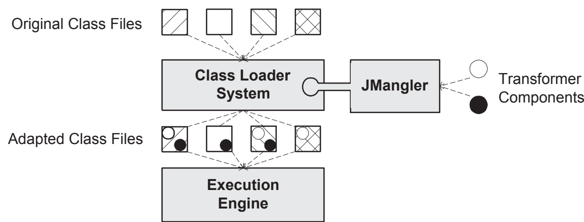

Figure 2: Architecture of the JMangler Framework

on target and decide which concrete transformation are to be carried out.

JMangler does not offer a dedicated transformation language, but instead requires analysis and transformations to be expressed in pure Java. For this reason, a good knowledge of Java's class file format is needed. However, we plan to develop a compiler for a higher-level aspect-oriented language (for example AspectJ) that compiles aspects into JMangler's transformer components.

The transformation process is partitioned into two phases. In the first phase, interface transformations are repeatedly executed in an arbitrary order until they stop to issue any further modification requests. In this phase, the framework checks the validity of requested transformations (with regard to binary compatibility), chooses the order in which transformations are to be applied, and carries out the concrete transformations.

In the second phase, only code transformations are executed. Each code transformation is executed exactly once, and the order in which the code transformations are applied must be defined by the composer of the transformer components. If repetition is needed, it must be expressed explicitly by the composer.

In order to be used as an interface transformer, a transformer component must implement the InterfaceTransformerComponent interface. Likewise, it must implement the CodeTransformerComponent interface in order to be used as a code transformer. It is also allowed to implement both interfaces in order to be used for both purposes. This does not affect the order of the two phases of transformation - a transformer component may play the role of an interface transformer only during the first phase and the role of a code transformer only during the second phase. However, in this way a transformer component is able to coordinate interface and code transformations on a detailed level.

JMangler's architecture is illustrated in figure 2. JMangler is configured by an XML file that includes information on the actual transformer components that are to be applied, and on the order of code transformations.

5 Example of Use

During the course of the TAILOR project, JMangler has been employed successfully for an implementation of LAVA, an extension of the Java Programming Language. In order to make the LAVA extensions effective for third party Java class files, special transformer components undertake the task of modifying their bytecode accordingly.

The ability to automatically combine interface transformers and apply them iteratively has proven to be an essential feature in the implementation of LAVA. It effectively allows JMangler to be used as a back end of the LAVA compiler. This is illustrated with the following, strongly simplified example.

One of the steps that LAVA takes to implement object-based inheritance is to automatically generate local forwarding methods for each method in the declared type of specially marked, so-called *delegatee* fields. For example in the following class, forwarding methods are generated for all methods that are included in class D's interface, since the declaration of field d includes the modifier delegatee.

```
public class C {
   public delegatee D d;

   // if method m is included in D,
   // delegatee (roughly) leads to
   // generation of the following method

   // public void m() { d.m(); }
}
```

A transformer component Forward is responsible for determination of the methods in the delgatee field type and the inclusion of appropriate forwarding methods in the class that contains the delegatee field. However, this process is complicated by the occurence of cyclic dependencies, as shown in figure 3.

In this example there are forwarding relations from D to C, from D to E, from E to D, and from E to F. If each class could be modified only once, in the first step, Forward would try to create forwarding methods for D. However, this would not create all necessary methods, since the methods that are "inherited" from F are missing in E. If Forward would first try to modify E, it essentially would face the same dilemma. This problem can be solved only by applying Forward repeatedly on each of the involved classes.

JMangler is able to deal with this kind of transformations. Since Forward is a pure interface transformation, there are no unwanted side effects resulting from interferences with transformer components that are responsible for other features of the LAVA language. For this reason, all interface transformers can be used without implicit interdependent knowledge.

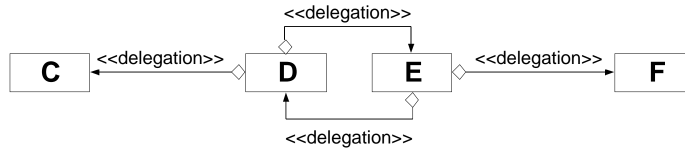

Figure 3: An example of a cyclic program structure that requires iteration of transformations.

6 Related Work

Some class libraries for class file representation and modification are available that can be used as a base for transformation tools, but they do not provide for sophisticated transformation processes. Advanced issues that are not dealt with by these libraries include the integration into the class loading and linking process, and a minimal amount of coordination of multiple transformers, at least.

JMangler uses the *Byte Code Engineering Library* (BCEL, former "JavaClass") [4]. Other libraries available are the *Jikes Bytecode Toolkite* (JikesBT) [5] and the *Bytecode Instrumenting Tool* (BIT) [6], as well as the API included in JOIE (see section 6.2).

6.1 Binary Component Adaptation

Binary Component Adaptation (BCA) [7] enables one to define modifications that are applied to classes during loadtime. For example, BCA enables new methods and fields to be introduced into a class, or interfaces to be extended by methods with standard implementations.

Whereas implementation of transformer components for JMangler require a good knowledge of Java bytecode, BCA allows one to express a predefined set of modifications in a Java-like syntax, which is easier to learn. They are compiled into so-called *delta files*, which can be used as parameters for invocations of the JVM.

On the other hand, JMangler allows for more complex modifications of class files and even allows dynamic considerations and dependencies between classes to be taken into account. BCA only allows for modifications of single classes at once based on static considerations.

BCA has been integrated tightly into an implementation of the Java Virtual Machine (as of JDK 1.1) for Sun Solaris. Therefore, it is more efficient than JMangler which is implemented in pure Java. For the same reason, BCA allows for modifications of system classes, which is not possible when JMangler is used. On the other hand, JMangler runs on top of all Java Virtual Machines that are compatible to JDK 1.3 and has been tested on Windows, Solaris and Linux.

6.2 Java Object Instrumentation Environment

The *Java Object Instrumentation Environment* (JOIE) [3] is another framework for loadtime transformations of Java classes. Transformations are declared as *transformers* that can be registered with a specialized class loader. This class loader creates object-based representations of class files immediately after they are loaded by the class loader. This representation is passed sequentially to each registered transformer. Since JOIE is implemented in pure Java it can be used with arbitrary implementations of the JVM.

JOIE is similar to JMangler to some degree. In fact, it has had a great influence on the design of JMangler. Nevertheless, there are some fundamental differences. Since JOIE employs a specially written class loader, it cannot deal with classes loaded by different class loaders, so it cannot modify applications that are in need of their own class loaders. JMangler is integrated on a deeper level of Java's core API, so it is able to modify all class files regardless of the class loader that they are being loaded by.

Furthermore, JOIE does not allow transformers to take dependencies between classes into account, since each transformer can be applied only to a single class file at once. Another fundamental difference is JOIE's lack of an advanced composition mechanism for independently developed transformers. Classes on target are passed just once to each registered transformer, so the problem of mutual triggers cannot be dealt with, contrary to what JMangler offers.

6.3 Javassist

Javassist is a class library for structural reflection in Java. It allows for modifications during loadtime and therefore can be compared to JMangler on a conceptual level.

Javassist does not provide a model of dealing with multiple transformers. Furthermore, only a strongly limited set of operations can be applied by Javassist. For example, it is not possible to introduce completely new defined methods into a class, but methods can just be copied from one existing class to another.

The integration into the linking process is implemented in a similar way like that of JOIE by providing a specialized class loader that creates an object-based representation of a class file. This results in the same limitation that is also present in JOIE. If an application needs to use its own class loader, its classes cannot be modified by Javassist.

7 Conclusion

So far, there is no satisfactory means in AOP systems to safely combine aspects that have been developed independently. Interferences between independently developed aspects must be controlled by a mediating component in order to avoid unwanted side effects. This fact greatly reduces the applicability of AOP systems. Especially in the context of component-oriented programming, where independent extensibility plays a central role [9], aspects themselves are hardly usable as units of composition in their current manifestations.

We have introduced the concept of partitioning transformations into interface and code transformations. This enables programmers to independently develop transformer components and have them automatically combined. In order to do so, there is no need to explicitly define the order of interface transformations. Only the order of code transformations has to be specified explicitly, since it is essential for the behavior of the resulting code.

Based on this concept, JMangler has been developed, a tool for loadtime transformations of Java classes. More detais on JMangler are given in [8].

Essentially we have shown that independent extensibility in the sense of component-oriented programming is feasible in the case of interface transformations. This result can be transferred easily to other AOP systems.

The question remains if more detailed criteria can be found for code transformations in order to achieve an even higher degree of independent extensibility. On the one hand it is conceivable that only a certain class of code transformations, which do not violate independent extensibility, is considered to be valid. Determination of the border between valid and invalid code transformations is the main challenge in this case. It may be important to take considerations into account that are drawn from data flow analysis or program slicing techniques.

On the other hand it will be interesting to explore the potential of attaching explicit specifications of composability properties to code transformer components. If these properties were verfiable automatically, they could be used as a basis for automatic composition.

The TAILOR project is supported by Deutsche Forschungsgemeinschaft (DFG) under grant CR 65/13.

References

[1] Aspect-Oriented Programming Home Page. http://www.parc.xerox.com/csl/projects/aop/

[2] Shigeru Chiba. *Load-Time Structural Reflection in Java*. in: Elisa Bertino (Ed.). *Proceedings of ECOOP2000* Springer LNCS 1850, 2000.

[3] Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. *Automatic program transformation with JOIE*. in: *Proceedings of the USENIX 1998 Annual Technical Conference*, Berkeley, USA, 1998. USENIX Association.

[4] Markus Dahm. *Byte Code Engineering Library*. http://bcel.sourceforge.net

[5] Chris Laffra. *Jikes Bytecode Toolkit*. http://www.alphaworks.ibm.com/tech/jikesbt.

[6] Han Bok Lee and Benjamin G. Zorn. *BIT: A tool for instrumenting Java bytecodes*. in: *USENIX Symposium on Internet Technologies and Systems Proceedings, Monterey, California, December 8–11, 1997*, Berkeley, CA, USA, 1997. USENIX.

[7] Ralph Keller, Urs Hölzle. *Binary Component Adaptation*. in: Eric Jul (Ed.). *ECOOP '98 – Object-Oriented Programming*. Conference Proceedings, Springer LNCS 1445, 1998.

[8] Günter Kniesel, Pascal Costanza, and Michael Austermann. *JMangler – A Framework for Load-Time Transformation of Java Class Files*. submitted for: *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001)*.

[9] Clemens Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[10] The Tailor Project. http://javalab.cs.uni-bonn.de/research/tailor/