JMangler – A Framework for Load-Time Transformation of Java Class Files

Günter Kniesel and Pascal Costanza
University of Bonn, Institute of Computer Science III
Römerstr. 164, D-53117 Bonn, Germany

{gk|costanza}@cs.uni-bonn.de

Michael Austermann
SCOOP GmbH, Im Ahlefeld 23
D-53819 Neunkirchen-Seelscheid, Germany

maustermann@scoop-gmbh.de

Abstract

Current proposals for load-time transformation of Java classes are either dependent on the use of a specific class loader or dependent on a specific JVM implementation. This is not due to an inadequacy of the Java platform but to the wrong choice of the level at which to hook into the Java Class Loader Architecture. JMangler follows a novel approach that ensures both class loader and JVM independence by hooking into the base class of all class loaders.

Furthermore, existing proposals do not allow transformers to be treated as components because implicit dependencies must be resolved manually. This paper shows that automatic composition is possible for the well-defined class of interface transformations *that still include powerful transformations, like addition of fields, methods and classes, and changes to the class hierarchy. Consequently interface transformers can be deployed jointly even if developed independently.*

1. Introduction

Tight development schedules and high quality expectations of customers create an ever increasing pressure for (re)use of readily available third-party components. At the same time, rapid changes in markets, legislation and enterprise strategies result in *unforeseen changes of requirements* that software has to meet. The *inability to adapt third-party components* whose source code is unavailable to unforeseen changes effectively prevents software development teams from achieving the necessary degree of reuse.

This dilemma led to the call for *information rich binaries*, which would contain enough symbolic information from the source program to enable automated analysis and transformation [8]. The Java Class File Format is the first commercially widely-adopted binary format that provides this property.

However, transformation of Java Class Files, while possible, is by no means easy to achieve. The foremost problem arises from the fact that Java allows classes to be loaded dynamically and the name of dynamically loaded classes to be determined at run-time, via reflection. Code that has been loaded already cannot be transformed anymore. So the only point where it is possible to determine all classes that are actually used by a program and adapt them as needed is the dynamic class loading process.

However, implementation of *load-time transformations of class files* is an intricate task. It requires intimate knowledge of the class file format and class loader architecture, at least. This motivated the development of *tools and frameworks for load-time transformation*, which provide different solutions for hooking into the class loading process. The next section summarizes the state of the art in load-time transformation of Java code, pointing out limitations of previous approaches that motivated this work. Section 3 presents the JMangler framework, the particular problems encountered in trying to overcome current limitations, and the solutions that we propose. An application example is given in section 4. JMangler is compared to related work in section 5. Section 6 summarizes the results and sketches future work.

2. State of the Art

Load-time transformation of Java class files is a relatively new research area. We are aware of only three approaches that go beyond the mere representation of Java class files[1] by providing complete solutions for the integration into the class loading and linking process of the Java platform. Other notable criteria for comparison are expressive power, and suitability for use in the software life cycle.

A short overview of each approach is given first, followed by a compilation of their main distinctions. It sheds light on the diverse forces that have to be taken into account in the design of a new approach and ultimately leads to the rationale behind JMangler.

2.1. Binary Component Adaptation

Binary Component Adaptation (BCA) [11] has been the first approach that enables compiled Java classes to be modified during load-time. Modifications are declared as *delta files* in a Java-like language that offers a predefined set of transformations. The invocation of the Java interpreter can be parameterized with one or more compiled delta files which are then applied to their target classes during load-time.

BCA has been integrated into the implementation of the Java Virtual Machine of JDK 1.1 for Solaris, and therefore cannot be used with other JVMs.

2.2. Java Object Instrumentation Environment

The *Java Object Instrumentation Environment* (JOIE) [3] is a Java framework for load-time transformations of class files. It does not employ a specific transformation language. Instead, transformations must be implemented as Java classes and registered with a specific class loader. This class loader creates object-based representations of class files during load-time and passes them sequentially to all registered transformers, which can perform arbitrary modifications.

Since JOIE is implemented in pure Java, it can be used with arbitrary implementations of the JVM.

2.3. Javassist

Javassist [2] is a class library for structural reflection during load-time that is implemented in pure Java. It takes a different route from those pursued by BCA and JOIE. It concentrates on the design of a meta-object protocol and regards its applicability during load-time as an implementation detail. Nevertheless, this approach effectively results in modifications of Java class files during load-time so it still fits into our category of related approaches.

The integration into the linking process is implemented in a way similar to JOIE, by providing a specialized class loader that creates an object-based representation of a class file. Structural reflection and modifications of classes are expressed programmatically by means of a dedicated API. Only a limited set of modifications can be applied to classes that adhere to Javassist's meta-object model.

2.4. Forces

From the analysis of these frameworks for load-time transformations, we have extracted the following forces that we have addressed in the implementation of JMangler. An overview of these forces is given in table 1.

2.4.1. Integration into the Java Platform

JVM independence BCA hooks into the class loading process via a modification of a JVM implementation, therefore tying itself to a specific platform and a specific JVM version. JOIE and Javassist achieve *JVM independence* by being implemented in pure Java. This is clearly preferable.

Applicability of transformers Java's class loader architecture [6] is an open system that allows for integration of arbitrary class loaders. The Java core API offers standard class loaders that are able to load classes from known sources, ranging from the local file system to remote web hosts via secure sockets.

A shortcoming of JOIE and Javassist is that they are *class loader dependent* – classes which need to be transformed are required to be loaded by a specific class loader that is supplied by the respective transformation framework. This makes them inapplicable to programs that need their own (different) class loader, as is the case for applets and distributed applications. The reason is that there is no way to let different class loaders simultaneously process the same copy of a class file.

It is clearly desirable to have *class loader independent* transformers, which are applicable without restrictions. Currently, BCA is the only approach that offers class loader independent transformers because of its integration into a specific JVM. The challenge is to find a way of intercepting all class loading activities but still remain JVM independent.

External configuration JOIE and Javassist require transformers to be programmatically registered with a specific class loader. This results in the need to recompile a program when the set of transformers needs to be changed. In contrast, BCA allows a set of delta files to be specified as

[1]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. JMangler uses the *Byte Code Engineering Library* (BCEL, former "JavaClass") [5]. Other libraries available are the *Jikes Bytecode Toolkit* (JikesBT) [9] and the *Bytecode Instrumenting Tool* (BIT) [10], as well as the API included in JOIE (see section 2.2).

	BCA	**JOIE**	**Javassist**	**Wish list**
Integration into the Java Platform				
JVM independence	–	√	√	√
Class loader independence	√	–	–	√
External configuration	√	–	–	√
Expressive Power				
Kinds of transformations	some	any	some	any
Preservation of binary compatibility	√	–	√	√
Multi-class transformers	–	–	(√)	√
Simple transformation language	√	–	–	√
Suitability for Component-Oriented Programming				
Multiple transformers	√	√	–	√
Independent extensibility	–	–	–	√

Table 1. State of the art of in load-time transformation

parameters to the invocation of the Java interpreter. This kind of external configuration is preferable in order to minimize turn-around time.

2.4.2. Expressive Power

Kinds of transformations BCA and Javassist place restrictions on the kinds of modifications that transformers are allowed to carry out. BCA allows classes and interfaces to be amended with new fields and methods, including code. It does not, however, allow existing members to be changed. Changes can be simulated by renaming existing members and adding new members with the old name. However, there is no way to have the code of the ”new“ methods programmatically generated from the code of the old methods. Javassist extends these options by allowing for a restricted set of changes to fields and methods, including changes to code. However, it limits the introduction of new methods to copies of existing ones.

In contrast, JOIE allows for all kinds of modifications, including addition, change and removal of fields and members as well as arbitrary changes to code. Of course, the possibility to change as many details as possible is preferable, but amounts to possibly breaking the expected properties and behavior of programs.

Preservation of binary compatibility For example, the Java Language Specification [7] defines a set of changes of a program that do not require clients to be recompiled. Such changes are said to preserve *binary compatibility*. A transformation system should support all changes that preserve binary compatibility and offer means to prevent all others.

Since Javassist only offers a strongly limited set of transformations to be carried out, binary compatibility is not an issue. BCA and JOIE allow for modifications that are complex enough to possibly break binary compatibility. Yet, JOIE does not provide any means to prevent violations of binary compatibility. BCA solves this problem by adding a specific attribute to every class that is compiled against an adapted component. When a name clash occurs, the system can resolve it by inspecting this attribute.

Multi-Class transformers BCA and JOIE only support *single-class transformers* which process classes one by one. Javassist additionally allows transformers to inspect properties of related classes in each step. Still, none of these approaches allows for *multi-class transformers*, which simultaneously transform multiple classes. Multi-class transformers are needed when mutual dependencies between classes occur during the transformation process. An example is given in section 4.

2.4.3. Suitability for Component-Oriented Programming

Transformers should be expressible in a form that takes the shape of components with explicit context dependencies only [12]. The aim is to make the transformation framework extensible by independently developed transformers. This is far from being trivial and has not been addressed explicitly by any of the previous approaches.

Multiple transformers The applicability of multiple transformers to the same classes is a minimal prerequisite for the use of transformers as components. Of the previous approaches, only BCA and JOIE support multiple transformers.

Independent extensibility In order to resolve *implicit dependencies* between transformers, BCA and JOIE burden the programmer with the specification of an order in which transformers are to be applied. Furthermore, when two or more transformers add properties to a class that cause the need of mutually reapplying the other transformers, it is hard — or even impossible — to find a reasonable order of transformations.

However, in order to allow transformers to be used as components, implicit dependencies must be avoided completely [12] and a support for determining an order of transformations is needed.

2.4.4. Simple Transformation Language

BCA is the only approach that offers a simple transformation language to express modifications of classes. It is easy to learn but only covers a limited set of modifications.

In fact, it is hard to design a clean and simple language that enables a wide range of relevant transformations, especially if arbitrary changes to all aspects of class members, including the code of methods, are to be supported. The design of a good high-level language model for transformations is an open research issue.

3. JMangler

JMangler is a Java framework for transformation of class files at load-time that extends the state of art with regard to almost all the aspects mentioned above. It plugs neatly into the Java architecture, providing an API for the creation of code and interface transformer components.[2] It further provides the ability to load sets of transformers, combine the transformations that they specify and perform these transformations on all classes of a program.

In the following sections we outline JMangler's basic concepts, explain how composition of independently developed transformers is achieved and describe how JMangler is integrated into the Java platform.

3.1. Basics

Legal Transformations JMangler supports all transformation of class files that do not violate binary compatibility. In particular, it supports

- addition of classes, interfaces, fields and methods,
- changes of a method's `throws` clause,
- changes of a class's `extends` clause that do not reduce the set of direct and indirect superclasses,
- changes of a class's `implements` clause that do not reduce the set of direct and indirect superinterfaces,

[2]In the following, the simpler term *transformer* is often used instead of transformer component.

- addition and changes of annotations that respect binary compatibility,
- changes of method code.

All transformations mentioned in the first five items of this list are called *interface transformations*. Changes of method code are called *code transformations*.

Contrary to fields, which are assigned default values by Java, 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. Consequently, the addition of a method including an initial method body is still regarded as a pure interface transformation.

Transformers Transformers are Java classes that implement specific interfaces (`InterfaceTransformer` and `CodeTransformer`). Implementation of these interfaces can be performed using JMangler's API. It supports three types of operations:

- analysis of class files, in order to determine whether a specific transformation is applicable,
- interface transformations and
- code transformations.

A transformer component that implements the operations of the `InterfaceTransformer` interface can perform one or many related interface tranformations. The same is true for code transformations. A transformer can play both roles by implementing both interfaces. Thus it is possible for one component to provide a consistent set of related interface and code transformations.

The `Counter` transformer, illustrated in Figure 1, is an example of such a combined transformer. It extends a class by a counter for each field. It further adds code to increment this counter prior to each direct read access to the associated field. The addition of counter fields is an interface transformation. The addition of instructions to increment counter fields is a code transformation. Nevertheless, both can be implemented in one transformer component and rely, for instance, on a common naming convention for the added fields.

3.2. Composition of Transformers

The main challenge in the design of JMangler resulted from the aim of enabling *unanticipated composition of independently developed transformers*.

When multiple transformers are active simultaneously, different transformers might be applicable to the same class. If their joint use has been anticipated, it is possible to explicitly specify how they are to be composed. However,

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


   public void manipulateB() {

      b.doSomething();
   }
}
```

Initial program

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

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

Result after application of `Counter` transformer

Figure 1. The `Counter` transformer

there is no satisfactory means to safely combine transformers that have been developed independently without their being specifically designed for joint use. Whenever independently developed transformers are provided as black boxes (which is the core idea of component oriented development), the composer most likely does not have enough knowledge of their design and implementation in order to decide on the proper composition.

In this context, the challenge in the design of JMangler was to find an *automatic* way of combining *black box* transformer components that *avoids unwanted side effects*. The problem has two main aspects, which are discussed in the following subsections:

- mutual triggering of transformers and
- order-dependent semantics of transformations.

3.2.1. Mutual Dependencies

The first problem of unanticipated composition is the possible occurence of mutual dependencies among transformers or among transformed classes.

Mutual Triggering A *property* of a program is a condition that refers to an individual class or to possibly complex relationships of many classes. A transformation that is performed if a certain property holds is said to be *triggered by that property*. For instance, "there is a public field in this class" is a property that might trigger the addition of accessor methods to a class. The property "there is a direct field access" might trigger the replacement of a field access instruction by an accessor method invocation.

A *transformation triggers another one* if it adds properties to a program that trigger the other transformation (directly or indirectly). A *transformer triggers another one* if some of its transformations trigger transformations of the other transformer. Two *transformers trigger themselves mutually* if each triggers the other one. Note that a transformer can also trigger itself.

Problem Neither the properties that trigger a transformation, nor the exact transformations triggered within a transformer, are known to the framework in advance. Therefore *the framework cannot determine whether a given set of transformers trigger themselves mutually*. It must always be prepared for the worst case, that is, for the occurence of mutual triggering.

Consequences In case of mutual triggering, applying each transformation only once may result in potentially incomplete programs, whose "gaps" can be filled only by repeated application of preceding transformations. Therefore *transformations must be iterated* until a fixed point is reached, that is until no tranformer requests any further changes.

As an example, consider another transformer, `Access`, which extends a class by access methods for each of its `public` fields and replaces all direct accesses to these fields by calls to the generated methods. The result of applying `Access` immediately after `Counter` is illustrated in Figure 2. The underlining on the right-hand side of the figure highlights code added by `Access` that requires a second application of `Counter`.

3.2.2. Order of Transformations

Unfortunately, iterating transformations that have been composed in different orders yields, in the general case, different fixpoints. This is illustrated in Figure 3.

The left-hand part of the figure continues the example from Figure 2 by repeatedly applying both transformers in the same order as before. This ensures that the `Counter` transformer is also applied to the methods generated by `Access` in the first iteration (the instruction `b_counter++` is inserted into the method `getB()`).

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








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

Result after application of `Counter` transformer

```
       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()    {
             return this.b;
          }

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

Result after application of `(1) Counter, (2) Access`

Figure 2. Applying transformations only once is not enough. The underlined code added by the `Access` transformer needs to be processed again by `Counter`

```
       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 3. State after end of second iteration. The results differ depending on the order of transformations.

The right-hand side shows the result of applying the two transformers in the *reversed order* (`Access`, `Counter`).

The difference in the final result manisfests itself in method `manipulateB()`. With the first ordering, the body of `manipulateB()` contains an instruction to increment the counter of field `b`. With the second ordering, the direct access to `b` is replaced by a call to the respective access method `getB()` *before* `Counter` is activated. Therefore, the instruction `b_counter++` is not inserted. The net effect is that `b_counter` is incremented twice as often in the left hand side version.

A programmer who compares the two results will immediately identify the right-hand result as the expected one and thus choose the corresponding ordering. However, the question at hand is how a suitable ordering could be chosen automatically by the framework without its having detailed knowledge about the particular transformers.

3.2.3. Partitioning of Transformations

Besides illustrating this negative result, the previous example is apt for another important observation, which leads to a partial solution. Whereas the resulting *method body* of `manipulateB` is dependent on the order of transformations, the *interfaces* of both versions of class `C` are exactly the same.

In fact, it can be shown that the result of a well-defined class of interface transformations is always independent of the order in which transformations are applied. In particular, interface transformations that are *positively triggered* and *monotone* provide the guarantee that terminating iterations produce a unique fixpoint.

Positive triggering A transformation is positively triggered, if it can be initiated by the existence of a particular property of a program, but must not be caused by the absence of such a property. This intuition is captured in the definition of "properties". A *property* of a program is any existentially quantified boolean expression that can be assembled *without using negation* in the language of JMangler's analysis API.

Without positive triggering, interface transformations would be order-dependent. Consider, for instance, a transformation T_{neg} that adds a method `m()` if a certain interface `I` is *not* implemented by the current class, and another transformation T that adds `I` to the list of implemented interfaces. If T is applied first, `m()` will *not* be added; otherwise it will.

Monotonicity In addition, sequences of interface transformations must be monotone. Intuitively, monotonicity means that transformations can add properties to a program but never remove properties. As in the case of negative triggering, it is easy to construct examples that show that non-monotonicity leads to order-dependence.

Note that the notion of "adding" refers to the semantics of the program, not its syntax. Replacement and removal of syntactic elements can still produce an *extended* program. For instance, a program with a method `public m()` is considered an extension of the same program with method `private m()`.

This example recalls the notion of binary compatibility. Indeed, *monotonic interface transformation sequences* are captured by a partial order on programs that mirrors Java's binary compatibility rules.

Partitioning With regard to our original problem statement, this section can be summed up as follows:

- Code transformations cannot be combined automatically (without human assistance) because their semantics are inherently order-dependent.
- Independent development and automatic combination (without human assistance) is possible for monotonic, positively triggered interface transformations.

This is the reason why JMangler partitions program transformations into code transformations and interface transformations, as described in section 3.1.

3.3. Transformer Configuration

A user who wants to transform a program at load-time can specify this easily in a configuration file. This file has a simple XML-based syntax describing

- the set of interface and code transformers to be applied,
- parameters to be passed to the transformers,
- the ordering of code transformers,
- and some other options (debugging, etc.).

Different application-specific transformers can be easily composed from the same set of basic transformers. Each composition specification can be stored in a different XML file. Switching between different configurations just requires providing a different file name as a parameter to the invocation of JMangler:

```
jmangler <configFile> <main> <parameters>
```

This invocation starts the JVM, loads JMangler and the transformers specified in the configuration file and then initiates execution of the program to be adapted. As the first step of the program's execution, the JVM attempts to load `main` which then triggers the transformation process on this class.

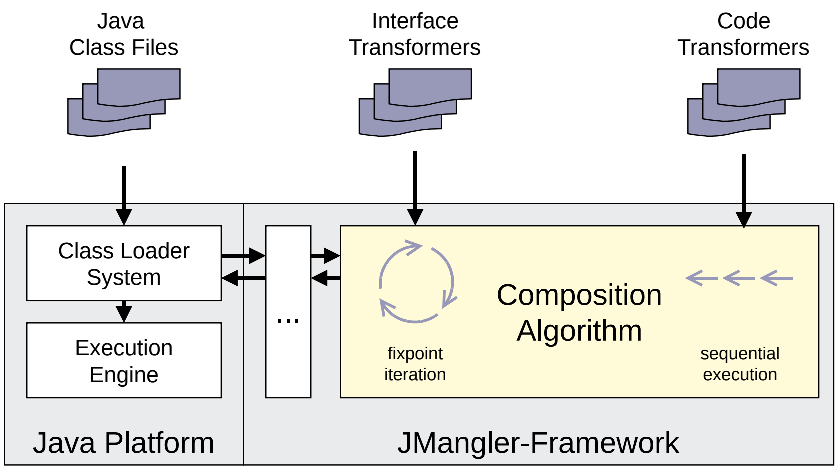

Figure 4. JMangler's Transformation Process

3.4. The Transformation Process

The transformation process is performed on each class that is loaded. When the transformation is complete, the transformed version of the class is passed to the execution engine of the JVM. It is also stored in a buffer of JMangler in order to be available for analysis by transformers of classes loaded later.

Multi-class transformers either find additional classes that they need to process in this buffer or they initiate loading of the yet unavailable classes. Thus JMangler always acts on two sets of classes: the classes that are (waiting for) being processed and the classes whose transformation has already been completed.

The distinction between interface and code transformations is reflected in the transformation process which is partitioned into two phases (see Figure 4). In the *first phase*, interface transformers are activated. Each interface transformer analyzes the classes under consideration, decides which transformations are to be carried out and requests these transformations from JMangler. The framework collects the transformation requests of all interface transformers, checks the validity of the requested transformations (with respect to binary compatibility), chooses the order in which legal transformations are to be applied, and performs the transformations. This process is repeated until no further interface modification requests are issued. If an illegal transformation is detected the process is aborted.

In the *second phase*, only code transformers are activated. They are executed exactly in the order indicated in the configuration file. If repetition is needed, it must be specified explicitly. Each code transformer analyzes the classes under consideration, decides which transformations are to be carried out and performs these transformations.

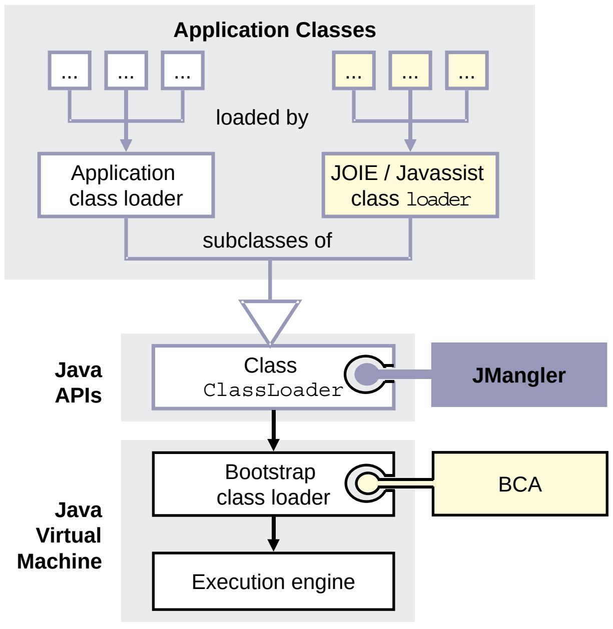

Figure 5. Three ways to hook into Java's Class Loader Architecture

3.5. Integration into Java's Class Loading Architecture

One of the main aims in the design of JMangler was to hook into the class loading system in a way independent of the class loader *and* the JVM.

Java's class loading mechanism is partitioned between the JVM and the Java APIs (Figure 5). On one hand, each platform-specific implementation of the

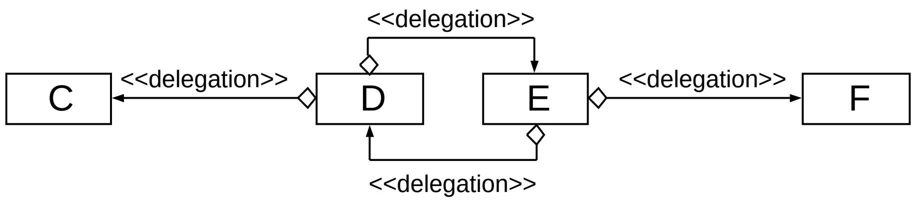

Figure 6. A cyclic program structure that requires iteration of interface transformations.

JVM contains a *bootstrap class loader*, which is responsible for loading system classes (that is, all classes that are part of the Java Development Kit and of standard extensions). On the other hand, the JDK contains the class `java.lang.ClassLoader` and subclasses thereof. `ClassLoader` is the common superclass of all class-loaders for application-specific classes. Programmers can customize the class loading system by writing their own subclasses of `ClassLoader`.

In our context, it is also important to understand that there is no way to let different class loaders simultaneously process the same copy of a class (file). This is the reason why applications that need their own specific class loader cannot be transformed by class loader dependent systems such as JOIE and Javassist.

The above summary of the class loading system is enough to explain two essential consequences. Any attempt to hook into the class loading system by creating an own subclass of class loader will result in a class loader dependent system. Any attempt to achieve class loader independence by modifying the bootstrap class loader necessarily compromises JVM independence.

JMangler achieves both goals by providing a modified version of the class `ClassLoader`. Because the modified behaviour is enforced for every subclass of `ClassLoader`, JMangler is activated whenever an application-specific class is loaded. However, JMangler still cannot transform system classes, which are loaded by the bootstrap class loader.

Figure 5 illustrates the different ways to hook into the class loading architecture.

4. Applications

During the course of the TAILOR project [13], 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 contents at load-time.

The ability to iterate positively triggered monotonic interface transformations until a fixpoint is reached and to modify multiple classes consistently during this process has proven to be an essential feature in the implementation of LAVA. It effectively allows JMangler to be used as a back end for the LAVA compiler. This is illustrated with the following, highly 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 accessible methods of the delegatee 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 6.

In this example, there are forwarding relations from D to C, from D to E, from E to D, and from E to F. Assume that `Forward` would try to create forwarding methods for D in the first step. However, this would not create all necessary methods, since the methods that are “inherited” from F are missing in E. If `Forward` were to first try to modify E, it would essentially face the same dilemma concerning the methods of C (not yet) “inherited” by D. This problem can be solved only by applying `Forward` repeatedly to each of the classes involved.

JMangler is able to deal with these types 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

	BCA	**JOIE**	**Javassist**	**JMangler**
Integration into the Java Platform				
JVM independence	–	√	√	√
Class loader independence	√	–	–	√
Transformation of system classes	√	–	–	–
External configuration	√	–	–	√
Expressive Power				
Interface transformations	√	√	√	√
Code transformations	–	√	(√)	√
Preservation of binary compatibility	√	–	√	√
Multi-class transformers	–	–	(√)	√
Simple transformation language	√	–	–	–
Suitability for Component-Oriented Programming				
Multiple transformers	√	√	–	√
Independent extensibility	–	–	–	√
Efficiency	√	?	?	–

Table 2. Comparison of JMangler to related approaches

interface transformers can be used without knowledge of implicit dependencies.

5. Related Work

In the following sections we summarize JMangler's features and compare them with the approaches that already have been described in section 2. The comparison is based on the forces that we have set forth in section 2.4, and on other characteristics that have been dealt with in the course of this paper. Please refer to table 2 for a summary of this comparison.

Integration into the Java platform Since JMangler is implemented in pure Java, it is, unlike BCA, *independent of a specific JVM*. Unlike JOIE and Javassist, it is *class loader independent* (section 3.5). However, it is not able to deal with classes that are loaded by the bootstrap class loader, since, for the sake of platform independence, this was not a viable design choice. While missing universality, JMangler is still significantly more general than an approach dependent on a class loader. BCA is the only universal approach.

JMangler offers an advanced concept of *external configuration* (section 3.3) that goes beyond the possibilities of BCA.

Expressive Power The goal of enabling a component-oriented approach for dealing with independently developed transformers has called for the distinction of two kinds of transformers. All approaches, including JMangler, provide for interface transformations. However, only JMangler is able to iteratively apply interface transformers in order to deal with mutual triggers. Arbitrary code transformations are provided by JOIE and JMangler only. Javassist allows for a restricted set of code transformations that adhere to its meta-object model.

JMangler offers support for *preservation of binary compatibility* (section 3.1). Unlike BCA and Javassist, it does so while still offering arbitrary modifications of Java classes.

Only JMangler is able to take mutual dependencies between classes into account and transform more than one class simultaneously by allowing for *multi-class transformers*.

JMangler does not provide a *simple transformation language*. Instead, we have focused on a general, API-based approach that can be made available quickly and provides all of the desirable properties of a transformation framework *except* for a simple syntax. Experiences provided by its deployment should help to fine-tune the system's design and implementation, leading to a stable and optimized basis for easy-to-use front-ends.

Suitability for Component-Oriented Programming BCA, JOIE and JMangler allow *multiple transformers* to be applied to the same set of Java classes. However, only JMangler is *independently extensible* by providing an advanced mechanism for dealing with independently developed interface transformers (section 3.2).

Efficiency JMangler spends significant time and space on maintaining representations of Java classes during runtime. BCA clearly has advantages in this area, because it can directly refer to representations of classes inside the JVM.

(There is no performance data reported in the literature on JOIE and Javassist.)

Note, however, that the overhead that JMangler incurs is only present during the transformation process. After transformation, applications run without any interference by JMangler – the transformer components may affect the efficiency of the actual classes, but this is outside of JMangler's scope.

6. Conclusions and Future Work

Previous approaches for load-time transformation of Java classes are either dependent on the use of a specific class loader or dependent on a specific JVM implementation. This is not due to an inadequacy of the Java platform but to a wrong choice of the level at which to hook into the Java Class Loader Architecture. JOIE and Javassist opt for supplying a class loader that is responsible for transformations, which excludes their application to many kinds of programs. BCA modifies a specific implementation of the JVM. JMangler follows a novel approach by modifying the base class of the Java class loader hierarchy, thereby enforcing transformations for classes that are loaded by arbitrary class loaders, except the bootstrap class loader. Therefore, JMangler cannot transform system classes.

Another important subject of this paper is the suitability of the framework for component-oriented programming. In previous approaches, there were no satisfactory means to safely combine transformers that had been independently developed. We have introduced the concept of partitioning transformations into interface and code transformations and have shown that automatic composition is possible for positively triggered, monotonic interface transformations. Corresponding transformers can be deployed jointly, even if developed independently. Only the order of code transformations has to be specified explicitly, since it is essential for the behavior of the resulting code. Criteria for automatic composability of code transformations are still an open issue.

Future work also needs to address JMangler's efficiency. For example, interface transformer components currently need to analyze classes in each iteration in order to determine if anything has changed. This overhead can be reduced by supplying detailed trigger events that are fired by specific types of modifications — for example addition of fields to a specific class. Interface transformer components are then only activated for the types of triggers for which they have registered.

Information on the current state of JMangler can be found at http://javalab.cs.uni-bonn.de/research/ jmangler/.

6.1. Acknowledgements

The authors thank Tom Arbuckle and the anonymous reviewers of the SCAM 2001 program committee for their critical comments. This work has been performed as part of the TAILOR project, directed by A. B. Cremers and 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: E. Bertino (Ed.). *ECOOP 2000 – Object-Oriented Programming* Proceedings, 313-336, 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*, New Orleans, Louisiana, USA, 1998. USENIX Association.

[4] Pascal Costanza, Günter Kniesel, and Michael Austermann. *Independent Extensibility for Aspect-Oriented Systems*. Accepted for: *Workshop on Advanced Separations of Concerns*. ECOOP 2001, Budapest, Hungary, 2001. Available at http://trese.cs.utwente.nl/Workshops/ecoop01asoc/

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

[6] Li Gong. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.

[7] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[8] Urs Hölzle. *Integrating Independently-Developed Components in Object-Oriented Languages*. in: O. M. Nierstrasz (Ed.). *ECOOP '93 – Object-Oriented Programming* Proceedings, 36-56, Springer, LNCS 707, 1993.

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

[10] 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.

[11] Ralph Keller and Urs Hölzle. *Binary Component Adaptation*. in: E. Jul (Ed.). *ECOOP '98 – Object-Oriented Programming*. Proceedings, 307-329, Springer LNCS 1445, 1998.

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

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