Altering Java Semantics via Bytecode Manipulation

Éric Tanter[1], Marc Ségura-Devillechaise[2], Jacques Noyé[2], and José Piquer[1]

[1] University of Chile, Computer Science Dept.
Avenida Blanco Encalada 2120, Santiago, Chile,
{etanter,jpiquer}@dcc.uchile.cl
[2] Ecole des Mines de Nantes, OCM group
La Chantrerie, 4, rue Alfred Kastler. B.P. 20722,
F-44307 Nantes Cedex 3, France,
{msegura,noye}@emn.fr

Abstract. Altering the semantics of programs has become of major interest. This is due to the necessity of *adapting* existing software, for instance to achieve interoperability between off-the-shelf components. A system allowing such alterations should operate at the bytecode level in order to preserve portability and to be useful for pieces of software whose source code is not available. Furthermore, working at the bytecode level should be done while keeping high-level abstractions so that it can be useful to a wide audience. In this paper, we present Jinline, a tool that operates at load time through bytecode manipulation. Jinline makes it possible to inline a method body before, after, or instead of occurrences of language mechanisms within a method. It provides appropriate high-level abstractions for fine-grained alterations while offering a good expressive power and a great ease of use.

1 Introduction

Altering the semantics of programs serves many objectives in software engineering, related to software *adaptation*. A particular case of software adaptation, highlighted by Keller and Hölzle in [1], is to make several off-the-shelf components interoperable [2]. To this end, Keller and Hölzle proposed binary component adaptation (BCA), a tool for performing coarse-grained alterations on component binaries. However, coarse-grained alterations, usually limited to modifications of the interface or of the type hierarchy, may turn out to be insufficient. Another objective addressed by alteration of program semantics is that of separation of concerns [3], as emphasized by the work carried out within the reflection community [4,5,6], and more recently, by the emerging paradigm of aspect-oriented programming (AOP) [7]. In both cases, an important objective is to separate the development of the functional core of an application from the implementation of its non-functional concerns, such as persistency, distribution, or security. The complete application is then obtained by merging the different parts together. Such a merging requires to perform fined-grained alterations

D. Batory, C. Consel, and W. Taha (Eds.): GPCE 2002, LNCS 2487, pp. 283–298, 2002.

within method bodies. The purpose of the work we present in this paper is to provide a tool enabling such alterations with the appropriate level of abstraction.

In Java, portable transformation mechanisms require code rewriting. This usually automated rewriting can be performed on source code or on bytecode. The Java community has already developed an impressive set of tools transforming source code: AspectJ [8] to support AOP, Sun's JavaScope project to instrument source code, a Dylan-like macro system called Java Syntactic Extender [9] and a class-based macro system, OpenJava [10]. Nevertheless, in many contexts, expecting source code availability is a mistake: off-the-shelf components usually ship in binary form, and sophisticated distributed systems, like mobile agent platforms, usually rely on dynamic class loading. Therefore, while still interesting in themselves, these tools are not generally applicable. This is why we claim that transformation tools should operate on bytecode.

Available transformation tools based on bytecode rewriting are usually inadequate for a wide and generic use. First, most of these tools offer bytecode-level abstractions. This is inadequate if the tool has to be used by a wide audience, since precise knowledge of the bytecode language is required. This point has been addressed by Javassist [11], which offers high-level abstractions. Though targeted to structural reflection, Javassist can be used to perform fine-grained alterations. However, in this domain, Javassist suffers from a limited expressive power and a lack of generality, as we will discuss in section 2.

In this perspective, we propose Jinline, a tool for altering Java semantics. Jinline operates on bytecode, keeps high-level abstractions, offers a good expressive power and generality. To summarize, Jinline makes it possible to inline a method body before, after, or instead of a language mechanism occurrence[1] within a method.

Traditionally, *inlining* means *replacing a call to a function by an instance of the function body* [12]. What Jinline actually does is inserting code or replacing code. The new code is defined by a method and therefore the inserted code is *conceptually* a method call, except that Jinline actually *inlines* this new method. Hence, although Jinline cannot be qualified as an inliner, most of its job consists of inlining pieces of code into others. In addition to this, Jinline provides two different sets of information:

1. **Static information at inlining time.** Jinline provides static information that can be used to drive the inlining process. For instance, in the case of a message send, it will provide the signature of the invoked method. This helps to decide whether inlining should occur or not, which method should be inlined and where (before, after, instead of).
2. **Dynamic information at run time.** Jinline ensures that the inlined method will receive as arguments *all the useful* dynamic information that

[1] By *language mechanisms* we refer to the standard mechanisms offered by the language, such as message sending, accessing fields, casting, etc. A *language mechanism occurrence* is a particular instance of a language mechanism in a piece of code.

can be extracted. This point is very important since it makes the tool particularly suited for implementing generic extensions, as we will exemplify in the rest of this paper. In the case of a message send, the dynamic information includes the method invoked, the method from which the invocation is done, references to the caller and the callee, in addition to the actual arguments of the invocation.

Applications of such an alteration tool are manifold. We have already mentioned the issue of off-the-shelf components integration. Two of the authors are actually working on an open implementation of a run-time MetaObject Protocol (MOP), Reflex [13]. Many transformers for the Reflex framework can be implemented with Jinline, thus increasing its expressiveness with caller-side interceptions. Jinline is also particularly adapted for implementing custom extensions and AOP systems.

The rest of this paper is organized as follows: in section 2, we will review the different Java bytecode manipulation tools and relate our work to them. In section 3 we will present Jinline, its interface to the outside world and an overview of its architecture. In section 4 we will present a simple example of applying Jinline. Section 5 will conclude the paper.

2 An Overview of Bytecode Manipulation Tools

One way of modifying a program is to alter its semantics by using reflection [14, 15]. However, the Java programming language does not provide support for altering the semantics of programs. Since the class model is closed (class `Class` and all the classes of the Reflection API are final), it is not possible to refine the semantics of language mechanisms by specializing the class model, as can be done in Smalltalk [16]. Therefore, alterations have to be implemented either at the virtual machine level, like in VM-based run-time metaobject protocols like Metaxa [17], Guaraná [18] and Iguana/J [19] thus sacrificing portability, or at the code level, through code transformation. We have already discarded the possibility of operating on source code for reasons of availability of the source code itself. This is why a number of propositions have been made to transform bytecode. These propositions differ in terms of the abstraction level of the entities a user is expected to program with, and in the expressive power or granularity of the transformations permitted.

2.1 Transformations Based on Bytecode-Level Abstractions

A number of extensions allow programmers to transform classes at load time at the expense of manipulating abstractions representing bytecode.

BIT [20] suffers from a too restricted scope: it only offers the possibility to insert before/after methods, but does not address transformation of interfaces or method bodies.

There are several general-purpose implementations of bytecode manipulation available: BCEL [21], JikesBT [22], and JOIE [23]. All of them translate the class file data structure into an intermediate representation, allow the user to perform modifications and to finally regenerate a valid class file data structure from the transformed intermediate representation. The bytecode-level API of Javassist [11] could fit into this category although bytecode instructions are not reified: the programmer is just provided with an iterator over a sequence of bytes. The main strength of these general-purpose extensions is their expressive power, since they are able to express anything that can be written in bytecode. However, their main drawback is to be low-level and therefore difficult to use.

2.2 Transformations Based on Source-Level Abstractions

Metaobject protocols (MOPs) are a natural framework for reifying high-level language entities [24]. Run-time MOPs are an approach to enable the run-time alteration of program semantics. Compared to static transformation systems – such as macro systems, inlining systems, and compile-time MOPs –, where the link between the modifier and the modified entity is merged at some point, run-time MOPs maintain this link, known as the *causal connection link* [14,15], at run time, thus enabling dynamic updates of this link at the expense of a certain overhead.

Reflex [13] and Kava [25] are run-time MOPs for Java that rely on load-time insertion of pieces of code (hooks) to transfer control to the metalevel at run time. These systems are bound to behavioral reflection, which is the ability of dynamically altering the behavior of objects. This approach is in fact complementary to static code transformation approaches in cases where dynamic adaptability or instance-specific alterations are needed (see for instance [26]).

BCA [1] is a bytecode modification tool with a high-level interface, but it only deals with external interfaces and class hierarchies, ignoring method bodies. Javassist [11] is a mature tool for load-time structural reflection in Java. Structural reflection is the ability of a program to alter the definitions of data structures such as classes and methods. With Javassist, the transformations that can be made are at the granularity of class or members. The main goal achieved by Javassist is a high-level and easy-to-use interface. To allow finer-grained transformations, Javassist has recently made public its bytecode-level API, which we mentioned in subsection 2.1. Recall that it lacks a concrete reification of bytecode instructions. To bridge the gap between its high-level and low-level APIs, Javassist offers a *code converter* to instrument method bodies through a high-level interface.

2.3 Limitations of the Code Converter of Javassist

The code converter of Javassist – the closest tool to our proposal – offers a simple high-level API to alter method bodies. This API allows inserting before/after methods, redirecting method invocations or field accesses, and replacing creations. We claim that its expressiveness is limited and that it lacks generality.

Its limited expressiveness is in fact not that much an issue since it can actually be upgraded, and also, in many cases, it is sufficient to alter such mechanisms as method invocations, field accesses and object creations. A more annoying problem is the limitation about the possible transformations: for instance, a field access can *only* be replaced by a static method call, and a method invocation can *only* be replaced by another method invocation on the *same* object with the *same* parameters.

But all in all, the major drawback of the code converter lies in the fact that it is not well-suited to designing generic solutions. Since Javassist lacks semantic information in the process of modifying bytecode (remember that Javassist does not reify bytecode instructions as such), the possible transformations are limited. The code converter does not perform any reification of *what* is actually occurring. For instance, an object creation can be replaced by a method invocation, but this method will not receive as argument the name of the class that was to be instantiated: it has to be *specific* to a type. This limitation is common to all transformations.

To illustrate this limitation, consider the following simple example: we want to set up a factory pattern [27] for instantiating any class in an existing application. That is to say, instead of calling directly `new`, we want to call a factory method. Designed with generality and extensibility in mind, the factory method would be:

```
public Object getInstance(String classname, Object[] args){...}
```

Then we want to transform all the instantiations so that they call this *unique* factory method, for instance:

$$\texttt{new Point}(1,2); \quad \Longrightarrow \quad \texttt{Factory.getInstance}(\text{"Point"},\ [1,2]);$$

This is not feasible with the code converter. The only possible replacement is:

$$\texttt{new Point}(1,2); \quad \Longrightarrow \quad \texttt{Factory.getPoint}(1,2);$$

The following issues come to light:

- First, the name of the instantiated class is not passed as a parameter, which implies that we need a method per class (a `getPoint` method, a `getTriangle` method, etc.).
- Second, the arguments are not packed, which means we need a method per set of parameters (a method `getPoint(int, int)`, another method `getPoint(Point)`, etc.).

It is easy to see that such an approach is not applicable to real world cases. What is needed is a tool that can systematically provide runtime information in a cost-effective manner to the new inserted code. In addition to this, more flexibility with respect to what code can be inserted is highly appreciable. This is exactly what Jinline is about.

3 Jinline

Jinline is a Java tool for altering the semantics of programs via bytecode manipulation. Jinline makes it possible to inline a piece of code before, after, or instead of occurrences of language mechanisms. Jinline provides all the necessary static information to drive the inlining process and, which is really important, it wraps at runtime all the available dynamic information and passes it to the inlined code. This makes generic alterations possible, unlike the `CodeConverter` of Javassist. We will show in section 4 how simple it is to solve with Jinline the issue presented in section 2.3.

Unlike low-level bytecode translators, Jinline keeps high-level abstraction while offering a good expressive power. This makes it usable by a wide audience and applicable to cases where source code is not available (e.g., components, mobile code systems). Inlining can be triggered on language-level mechanisms (message send, constructor send, cast, ...) and the inlined code is also expressed in source code.

Jinline is integrated within the Javassist framework for load-time bytecode transformation which was designed with type safety[2] and correctness[3] in mind. Types and methods reifications in Jinline are those of the core Javassist API.

3.1 A User View on Jinline

Inlined code. A natural formalism for representing a piece of code to inline is indeed a *method*. The programmer is responsible for writing it in Java and compiling it using a standard compiler. It will then need to feed the inliner with the bytecode definition of this method. In addition to this, choosing to inline methods provides us with a natural way to pass dynamic information at run time to the inlined piece of code: all relevant information is packed and passed as argument of the inlined method. We will describe the structure of this information later in this section.

When a method is inlined, it is not recursively processed by Jinline. This is to avoid inconsistencies: if one wants to inline a method for method invocations and another method for casts, Jinline will actually perform both transformations simultaneously. Therefore the order of the transformations does not have any impact on the result: none of the inlined methods will be transformed. Nevertheless, it is not impossible to imagine cases where one would like to process the inlined code. This can be done by completing the first process and then performing a second pass over the transformed code.

Descriptions. All the language mechanisms are represented in Jinline by *description classes*. There is a description class for message send, one for cast, etc. An instance of such a class contains all the static information that describes the occurrence of the mechanism. The methods of each class are the accessors to

[2] You cannot get an invalid reference to a reification of a class or method.

[3] Javassist greatly limits the possibility of producing bytecode rejected by the JVM.

the static information that is provided to the user for driving the inlining process. Figure 1 shows the hierarchy of description classes in Jinline. Jinline does not cover message reception, since it can typically be implemented by coarser-grained transformations (as done in the reflection package of Javassist). Also, Jinline does not cover control structures, since they are not always represented the same way in the bytecode that in the source code: some compiler optimizations may eliminate or alter control structures existing in the source code.

One can refer to a language mechanism by using its description class object. For instance, `MessageSend.class` represents the language mechanism of sending messages, while an instance of this class represents an occurrence of this mechanism.

The process. The inlining process is based on a listener-notifier schema as illustrated in Figure 2. In Jinline, method definitions are the unit of transformations. When parsing a method to alter, Jinline notifies a listener each time it encounters an occurrence of interest. The notification embodies all the static information that can be extracted about the occurrence. The listener can then analyze this static information and decide to inline or not a piece of code before, after, or instead of the occurrence.

An inlining process is represented by an instance of the `Jinliner` class. The inlining process is specified by attaching the proper listener to each language mechanism of interest. For instance, to specify that the `Jinliner` object should send a notification upon occurrences of message send, one should write:

```
jinliner.notifyUpon(MessageSend.class, aJinler);
```

One of the advantage of such an approach is that it is possible to specify a common super type of mechanism in the specification. For instance:

```
jinliner.notifyUpon(Description.class, aJinler);
```

implies that `aJinler` will be notified of any occurrence of a known mechanism.

To instrument a method, this method first needs to be reified into a Javassist `CtMethod` object:

```
CtMethod myMethod = aCtClass.getDeclaredMethod("myMethod");
```

Then the process can be started:

```
inliner.process(myMethod);
```

It is also possible to process an entire class (by passing a `CtClass` object to `process`) in which case, by default, all its public methods and constructors will be processed.

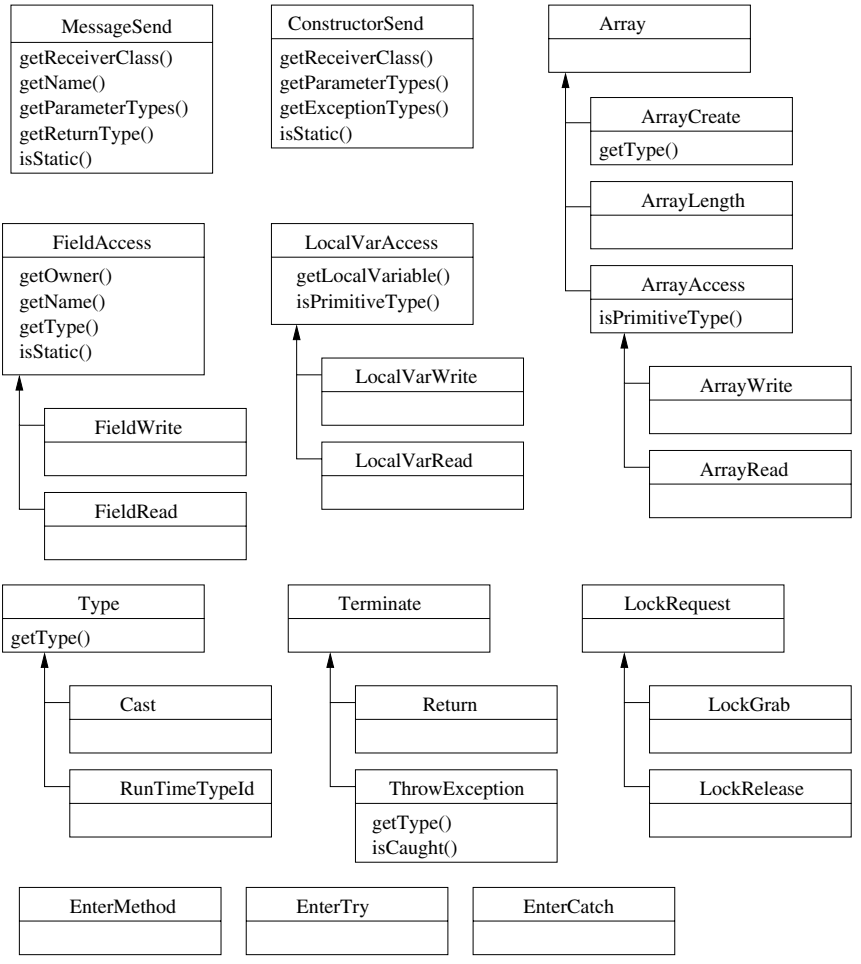

Fig. 1. The hierarchy of description classes.Note that the common superclass `Description` has been omitted for the sake of readability.

Jinlers and static information. Jinlers are objects that listen for notification from the inliner and that drive the inlining process. Such objects implement the `Jinler` interface, which declares the method:

```
public ToInline notify(Description desc, Context context);
```

This means that upon notification, the `Jinler` object receives an instance of a description class, which encapsulates all the static information about the occurrence of the mechanism. For instance, to reason about the return type of a message send, one can use:

```
if(((MessageSend)desc).getReturnType().equals(...)) ...
```

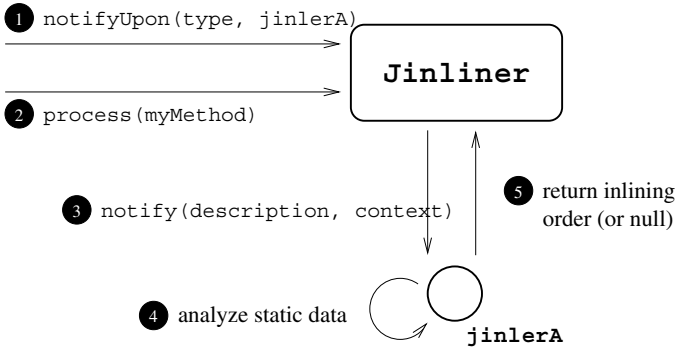

Fig. 2. The inlining process is based on a notification scheme. (1) *jinlerA* is registered for notification of occurrences of *type*. (2) The Jinliner is given the method to process. (3) Each time an occurrence of *type* is encountered, the Jinliner notifies *jinlerA*. (4) *jinlerA* analyzes the static data about the occurrence. (5) It returns an inline order or null if no inlining should take place.

The `Context` object given upon notification is simply an object that encapsulates the current class and the current method being processed.

Finally, to specify the desired inlining (that is, which method(s) to inline and where), a `Jinler` returns a `ToInline` object. Such an object is a structure of `CtMethod` objects to inline before, after, or instead of the occurrence. For instance:

```
return new ToInline(myMethod);
```

specifies that the inliner should inline `myMethod` *instead of* the current occurrence, whereas:

```
(1) return new ToInline(beforeMethod, afterMethod);
(2) return new ToInline(beforeMethod, null);
```

(1) specifies that `beforeMethod` should be inlined *before* the occurrence and that `afterMethod` should be inlined *after* it, and (2) specifies that only `beforeMethod` should be inlined *before* the occurrence.

Dynamic information and inlinable methods. At run time, dynamic information is packed and passed to the inlined method. Great attention has been paid to provide to the inlined method all the information available. The information is passed as an array of objects containing only standard objects and arrays, in order to be cost-effective. Table 1 shows how the information is ordered in the given array depending on the mechanism.

The way in which dynamic information is passed imposes a compatibility rule over inlinable methods. In Jinline, to be inlinable, a method has to accept either an array of objects as unique parameter or no parameter at all. If the

Table 1. Information delivered at runtime to the inlined method.The information is passed as an `Object[]`. For all mechanisms, the first two elements are the same: the method in which the occurence is located and the instance that is involved (*this*). The table shows how specific information is stored in the array for each language mechanism. (*) Due to restrictions in the JVM specifications, only before/after insertion is possible.

Language mechanism	**Dynamic information passed to the inlined method**	**Index: type**	**Expected return value**
Common	qualified name of altered method	0: `String`	
	this object	1: `Object`	
Message sent	qualified name of invoked method	2: `String`	invocation result
	target instance	3: `Object`	
	values of arguments	4: `Object[]`	
Constructor	qualified name of target class	2: `String`	object
	argument values	3: `Object[]`	
Cast and RTTI	qualified name of target type	2: `String`	object (cast) or boolean (RTTI)
	target instance	3: `Object`	
Field read	qualified name of field read	2: `String`	field value
	target instance	3: `Object`	
Field write	qualified name of field written	2: `String`	void
	target instance	3: `Object`	
	new field value	4: `Object`	
Local variable read	is this	2: `Boolean`	variable value
	is parameter	3: `Boolean`	
	is primitive type	4: `Boolean`	
Local variable write	is this	2: `Boolean`	void
	is parameter	3: `Boolean`	
	is primitive type	4: `Boolean`	
	new variable value	5: `Object`	
Array creation	qualified name of array type	2: `String`	array
Array length	array whose length is being accessed	2: `Object[]`	array length
Array read	array which is being accessed	2: `Object[]`	object
	index of the array element	3: `Integer`	
Array write	array which is being accessed	2: `Object[]`	void
	index of the array element	3: `Integer`	
	new array element value	4: `Object`	
Lock grab (*)	target instance	2: `Object`	void
Lock release (*)	target instance	2: `Object`	void
Throw exception	exception instance	2: `Throwable`	exception to throw
Return	value that is to be returned	2: `Object`	object to return
Enter a method or a try/catch	*none*		void

method does not accept any parameter, no dynamic information will be passed (which is much more efficient when dynamic information is not needed). This entails that when creating a `ToInline` object, the `CtMethod` objects given to the

constructor must fulfill this compatibility requirement otherwise an exception is thrown.

As far as return values are concerned, no check is made when creating a `ToInline` object. In case of inlining before/after, the return value is simply not taken into account since it is irrelevant. In the case of a replacement, it is up to the programmer to guarantee that the value returned by the inlined method is compatible with the expected type (see table 1). Jinline only takes care of wrapping and unwrapping primitive types and exceptions, which is actually the only thing it can do systematically. At run time, if an inlined method returns an incompatible value, a `ClassCastException` is thrown.

3.2 Overview of Implementation

We have created our own reification of bytecode instructions and created a high-level symbolic object for bytecode modification: `BytecodeSequence`. Such objects represent sequences of bytecodes and offer services to manipulate them, in particular inserting another sequence at a given index. Since our main goal is easy insertion of sequences into others, our reification of bytecode is a straightforward one. We are not interested in optimizing the bytecode or performing analyses on it. This is why we do not change the bytecode format, as is done in a dedicated framework like Soot [28].

A `BytecodeSequence` object can be created out of Javassist `MethodInfo` objects, which are the low-level reifications of methods. During parsing, any information (indexes, jumps, local variables, etc.) is translated into symbolic data. Manipulation over bytecode sequences is all symbolic. This makes copies of method bodies into others fairly straightforward. It is only at generation time that the symbolic information is translated back to raw data. The `MethodInfo` object is then updated with this new raw data.

For the inlining part, a `MethodParser` is responsible for parsing a method body and notifying the appropriate Jinlers whenever needed. When an inlining needs to be done, it is delegated to a `MethodInliner`.

A `MethodInliner` is responsible for inlining the method in a semantically correct manner:

- if the inlined method expects dynamic information, it first inserts a prologue that does the wrapping of all the parameters. Such a prologue is specific to each supported mechanism. The array of objects that the inlined method expects as an argument is transmitted via an extra local variable;
- it adjusts the method to inline to remove its return statements and ensure stack consistency. This modification is common to all mechanisms. Note that the value returned by the inlined method is also transmitted via a local variable;
- it appends the adjusted method body after the prologue;
- it adds an epilogue that unwraps the return value and manages the exceptions that the inlined code may throw. More precisely, if the inlined method throws an exception that was expected by the original code, it is simply

thrown again, otherwise an `UndeclaredThrowableException` is thrown, as is done in the Dynamic Proxy API of Java [29]. Such an epilogue is common to all mechanisms as far as the return value is concerned, but the exception handling part is specific to each mechanism. Recall that in the case of a before/after inlining, the return value is simply ignored.

Figure 3 below illustrates the inlining operation in the case of a replacement.

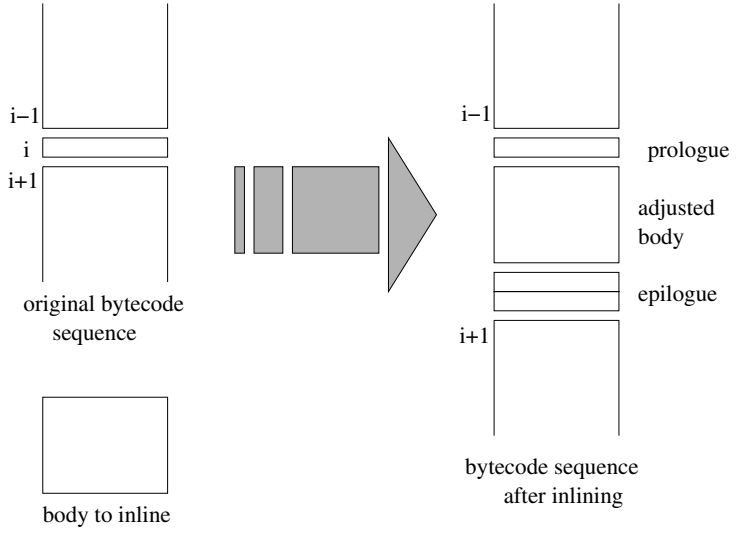

Fig. 3. The inlining operation at the bytecode level (replacement of i).The method body to inline is ajusted before insertion. The prologue wraps the arguments to the inlined body (if needed). The epilogue unwraps the return value (if needed) and handles exceptions.

4 Example

Let us come back to the example presented in section 2.3 when highlighting the limitations of the `CodeConverter` of Javassist. This section clearly shows how generic solutions can be implemented easily thanks to Jinline's ability to provide dynamic information. The objective is to use Jinline to replace any instantiation statement by a call to a generic factory method. We therefore assume that we have developed the factory:

```
public class Factory {
  public Object getInstance(String classname, Object[] args)
  { // factory code }
}
```

We now present the 3-step process used to solve the problem with Jinline.

1. Code to inline. The first step is to write a method that contains the code to inline. This is done by defining a sample class:

```
  public class FactorySample {
    public Object newMethod(Object[] jinArgs){
 (1)  String classname = (String) jinArgs[2];
 (2)  Object[] args = (Object[]) jinArgs[3];
 (3)  return Factory.getInstance(classname, args);
    }
  }
```

Since in this example we make use of the dynamic information that Jinline provides to the inlined code, the method we will inline, `newMethod`, accepts as unique argument an array of objects. According to the way dynamic information is structured (see table 1), we know that the third argument in the array is the name of the class and that the fourth argument is the array of arguments to the constructor. Hence, we first retrieve those values (1 & 2). Then, we simply invoke the factory method with the extracted values (3).

This sample class has to be compiled so that it is possible to obtain a Javassist reification of the method `newMethod`.

2. The Jinler. Next, one needs to define the Jinler, that is to say, the entity responsible for driving the inlining process:

```
  public class FactoryJinler implements Jinler {
    CtMethod newMethod;

    FactoryJinler(){
(1)   newMethod =
        ClassPool.getDefault().get("FactorySample")
                              .getDeclaredMethod("newMethod");
    }

    public ToInline notify(Description desc, Context cont){
(2)   if(desc instanceof ConstructorSend)
(3)     return new ToInline(newMethod);
(4)   return null;
    }
```

In the constructor, the Javassist reification of `newMethod` is obtained (1). In `notify`, the description object is filtered according to its type (2). If it corresponds to an instantiation, then we return an inline order, specifying that we want to inline `newMethod` instead of the current instantiation (3). Otherwise, we specify that no inlining should occur (4).

3. Connecting to the Javassist framework. The last step is to connect to the Javassist framework for load-time bytecode transformation. For the readers which are not familiar with Javassist, let us recall that it is possible to define a `Translator` object that is notified each time a class is loaded and that can perform some transformation before the class is actually loaded. Here is a translator for the Javassist framework that uses a Jinliner and the Jinler defined above:

```
public class FactoryTranslator implements javassist.Translator {
  Jinliner inliner = new Jinliner();
  Jinler jinler = new FactoryJinler();

  public void start(ClassPool pool){
(1)  inliner.notifyUpon(ConstructorSend.class, jinler);
  }

  public void onWrite(ClassPool pool, String classname){
(2)  CtClass clazz = pool.get(classname);
(3)  inliner.process(clazz);
  }
}
```

When connected to the framework, the translator is initialized by a call to its `start` method. The initialization work in this case simply consists of telling the Jinliner that it should notify the Jinler upon occurrences of constructor sends (1). Then, each time a class is about to be loaded, the translator is informed by an invocation of its `onWrite` method. Here, we simply get a reification of the class (2), and pass it to the Jinliner (3).

With this straightforward code, we are able, at load time, to systematically transform any occurence of instantiation statements by the appropriate call to our generic factory.

5 Conclusion

In this paper we have presented Jinline, a tool for altering Java semantics via bytecode manipulation at load time. Jinline allows fine-grained alterations on bytecode while keeping high-level abstractions. With Jinline, a method body can be inlined before, after, or instead of any language mechanism occurrence within a method. Static information is given to the programmer to drive the inlining process and dynamic information can be passed to the inlined method, making generic alterations possible.

The main achievement of Jinline is to be a simple and powerful tool, with a wide set of possible applications, that seamlessly fits within the Javassist framework to extend the range of easily accessible bytecode manipulation in Java.

With regards to future work, we plan to perform some benchmarks and focus on optimizing the bytecode generated by Jinline, since code explosion might become an issue in cases where many alterations are performed within the same

method. Possible tracks are the use of subroutines to factor out common parts and basic optimizations on the generated bytecode sequences. As for applications, Jinline will be applied to build a comprehensive library of transformers for the Reflex framework, an open implementation of a run-time MOP [13].

Acknowledgments. We would like to deeply thank Shigeru Chiba for the highly valuable remarks he made on the first version of this paper. We are also grateful to the anonymous reviewers for their comments.

This work was partially funded by Millenium Nucleous Center for Web Research, Grant P01-029-F, Mideplan, Chile.

References

[1] Keller, R., Hölzle, U.: Binary component adaptation. In: Proceedings of ECOOP'98 - 12th European Conference on Object-Oriented Programming, Brussels, Belgium, Springer-Verlag (1998) 307–29
[2] Wegner, P.: Interoperability. ACM Computing Surveys **28 (1)** (1996)
[3] Tarr, P.L., Ossher, H., Harrison, W.H., Jr., S.M.S.: N degrees of separation: Multi-dimensional separation of concerns. In: International Conference on Software Engineering. (1999) 107–119
[4] Stroud, R.J., Wu, Z.: Using Metaobject Protocols to Satisfy Non-Functional Requirements. In: Advances in Object-Oriented Metalevel Architectures and Reflection. CRC Press (1996) 31–52
[5] Tanter, E., Piquer, J.: Managing references upon object migration: applying separation of concerns. In: Proceedings of the XXI International Conference of the Chilean Computer Science Society (SCCC 2001), Punta Arenas, Chile, IEEE Computer Society (2001) 264–272
[6] McAffer, J.: Meta-level architecture support for distributed objects. In: International Workshop on Object-Orientation in Operating Systems (IWOOS'95). (1995)
[7] Kiczales, G., Irwin, J., Lamping, J., Loingtier, J., Lopes, C., Maeda, C., Mendhekar, A.: Aspect Oriented Programming. In: Special Issues in Object-Oriented Programming, Max Muehlhaeuser (general editor) et al. (1996)
[8] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An overview of AspectJ. Proceedings of ECOOP 2001 (2001)
[9] Bachrach, J., Playford, K.: The Java syntactic extender. OOPSLA 2001 conference proceedings (2001) 31–42
[10] Tatsubori, M.: An extension mechanism for the Java language. Master's thesis, Tsukuba, Japan (1999)
[11] Chiba, S.: Load-time structural reflection in Java. European Conference on Object-Oriented Programming (ECOOP'00) (2000)
[12] Howe, D.: FOLDOC: Free On-Line Dictionary Of Computing. (1993) `http://foldoc.doc.ic.ac.uk`.
[13] Tanter, E., Bouraqadi, N., Noyé, J.: Reflex – Towards an Open Reflective Extension of Java. In: Proceedings of the Third International Conference on Metalevel Architectures and Advanced Separation of Concerns (Reflection 2001). Volume 2192 of Lecture Notes in Computer Science., Kyoto, Japan, Springer-Verlag (2001) 25–43

[14] Smith, B.: Reflection and semantics in Lisp. In: Proceedings of the 14th Annual ACM Symposium on principles of programming languages, POPL'84. (1984) 23–25
[15] Maes, P.: Computional reflection. PhD thesis, Artificial intelligence laboratory, Vrije Universiteit, Brussels, Belgium (1987)
[16] Goldberg, A., Robson, D.: Smalltalk-80: The Language and its Implementation. Addison-Wesley (1983)
[17] Golm, M.: Design and implementation of a meta architecture for Java. Master's thesis, Leipzig Germany (1997)
[18] Oliva, A., Calciolari Garcia, I., Buzato, L.: The reflexive architecture of Guaraná. Technical report, IC-98-14, Institute of Computing, State University of Campinas (1998)
[19] Redmond, B., Cahill, V.: Supporting Unanticipated Dynamic Adaptation of Application Behavior. In: Proceedings of ECOOP 2002. Volume 2374 of Lecture Notes in Computer Science., Málaga, Spain, Springer-Verlag (2002) 205–230
[20] Lee, H.B., Zorn, B.G.: BIT: A tool for instrumenting Java bytecodes. In: USENIX Symposium on Internet Technologies and Systems. (1997)
[21] Dahm, M.: Byte code engineering. In Cap, C., ed.: Proceedings of JIT'99, Berlin. (1999) 267–277
[22] AlphaWorks: JikesBT. `http://www.alphaworks.ibm.com/tech/jikesbt` (1998)
[23] Cohen, G., Chase, J., Kaminsky, D.: Automatic program transformation with JOIE. in Proceedings of the 1998 USENIX Annual Technical Symposium (1998)
[24] Kiczales, G., Des Rivières, J., Bobrow, D.: The Art of the Meta-Object Protocol. MIT Press (1991)
[25] Welch, I., Stroud, R.: Kava - a reflective Java based on bytecode rewriting. In: 1st OOPSLA Workshop on Reflection and Software Engineering (OORaSE'99). Volume 1826 of Lecture Notes in Computer Science., Denver, USA, Springer-Verlag (2000) 165–167
[26] Tanter, E., Vernaillen, M., Piquer, J.: Towards Transparent Adaptation of Migration Policies. In: 8th ECOOP Workshop on Mobile Object Systems (EWMOS 2002), Málaga, Spain (2002)
[27] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing Series. Addison-Wesley (1994)
[28] Vallee-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java optimization framework. In: Proceedings of CASCON 1999. (1999) 125–135
[29] SUN Microsystems: Dynamic Proxy Classes. (1999) `http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html`.