Pluggable Reflection: Decoupling Meta-Interface and Implementation

David H. Lorenz* and John Vlissides†

Abstract

Reflection remains a second-class citizen in current programming models, where it's assumed to be imperative and tightly bound to its implementation. In contrast, most object-oriented APIs allow interfaces to vary independently of their implementations. Components take this separation a step further by describing unforeseeable attributes—the key to pluggable third-party components. This paper describes how reflection can benefit from a similar evolutionary path.

1 Introduction

A reflective computation [19] is one that has an accurate representation of itself, and the computation is consistent with its representation. A change to the computation appears in the representation and vice versa. Many object-oriented languages (OOLs) provide a mechanism—*reflection*—that supports reflective computation explicitly [32][33]. Reflection is a versatile tool, applicable at run-time, compile-time, load-time, and during component assembly.

A tenet of software development in general and object-oriented design in particular is to program to an interface. Doing so allows client and implementation to evolve separately, thereby facilitating code evolution, portability, and testability. Maintaining several native implementations for the same interface also promotes portability, much as multiple back-ends do for a compiler. Swappable implementations can even simplify prototyping and testing, say, by providing a choice of instrumented, optimized, and stubbed-out implementations.

Unfortunately, no mainstream OOL extends the benefits of programming to an interface to its reflection facilities. Code that uses reflection is assumed to access the reflective representation—they are one and the same. The result is a situation resembling the one that gave rise to printer drivers. Back then, each application was tailored to work with specific printers. Buying a printer with new or different features required upgrading or replacing the application. Something similar can be said of current applications that use meta-information.

The problem lies in a strong coupling between the reflection interface and its implementation. A debugger, for example, may obtain information about a program being debugged either from a source code repository or through reflection. Debuggers usually work with one information source or the other, not both; changing the source would require substantial rework of the debugger. But that needn't be the case. Reflection is just another application programming interface (API) that can have several implementations.

Consider how a printer driver defines a standard API through which printer and application communicate. Any application that can print may use the print driver's interface, confident that it will work subsequently with new printers without change. Conversely, a printer needn't be aware of the applications it serves; it can rely on the driver to relay status and problem information to applications. In the same way, decoupling the reflection interface from its implementation affords the novelty of more than one implementation of meta-information. Then the debugger, for example, could be plugged by third parties into either a source code repository or reflection. We call this vision *pluggable reflection*.

The paper illustrates two extralingual approaches to pluggable reflection. In the first, clients of reflection are structured as visitors of objects representing a grammar (i.e., an instance of the INTERPRETER pattern) [7]. For added flexibility, the second approach implements the INTERPRETER pattern dynamically rather than statically using a set of event-based ReflectionBeans components.

Both examples are based on an abstract syntax representation of reflection. Programming against a grammar as opposed to a physical implementation of reflection is key to retargeting clients of meta-information to different sources. In fact, the INTERPRETER pattern's class hierarchy can mimic Java's reflection facilities precisely enough that clients can use one or the other just by changing `import` statements.

The ability to retarget tools to different meta-information sources is not only useful but sometimes necessary. In Section 2 we motivate this work by explaining the tradeoffs between two meta-information sources. In Section 3 we show the benefits of choosing the most appropriate source, and then the benefits of using both. We describe our experience implementing a *javadoc*-like tool that can be easily retargeted to various reflective sources. An alternative implementation strategy applies the object-to-component transformation that we present elsewhere [18] to produce JavaBeans from Java's Reflection API.

2 Motivation

The need for pluggable reflection arises as soon as you have more than one source of meta-information. Consider for example the overlap between reflective facilities and source code repositories. Both manage information about classes and their relationships, and both aid in the software development process—they can make changing, tuning, and debugging your program a lot easier.

* College of Computer Science, Northeastern University, Boston MA 02115 USA.
† IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598 USA.

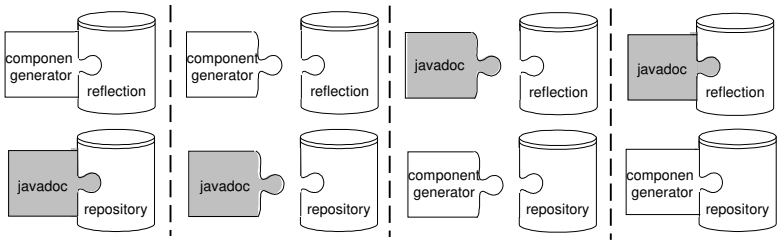

Figure 2: Pluggable reflection

Yet reflection and repositories have obvious differences that complicate interoperability. Ideally we could ignore such differences, choose either approach, and switch between them freely; the origin of reflective information would be transparent to clients. But in practice, the reflection interface and its implementation are strongly coupled. That is, code that *uses* reflective information and code that *accesses* the reflective representation are one and the same.

Suppose you are writing a tool that processes source code, such as a class browser. It requires services found in both the reflection interface and the class repository interface at your disposal (that is, the intersection of those interfaces). You therefore have a choice—you can get class information from either source. Your browser can compile and load classes and then use reflection to reveal their structure, or it can parse the class source, store the information in the repository, and query the repository for the same information (Figure 1).

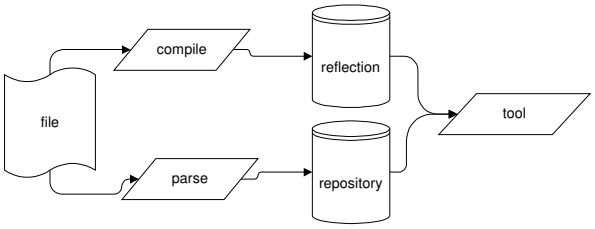

Figure 1: The reflection-repository duality

Neither approach is adequate in all circumstances. Here are several trade-offs to consider:

1. *Completeness*. A reflection-based approach precludes browsing code that is not compilable. Repositories only require the code to be reasonably parsable, and they can make broad assumptions about incomplete portions.
2. *Coupling.* Reflection is closely associated with the programming language. As a result, there is a tight binding between reflection facilities, client code, and the run-time environment. A repository, in contrast, can be more loosely coupled to its clients. One result is that testing a client apart from the repository is generally easier than testing it apart from the reflection machinery.
3. *Semantic mismatch*. A repository may have facilities beyond reflection, such as versioning and concurrency control, that are overkill for simple clients like the class browser, which has no editing or versioning capabilities. The repository doesn't justify its cost for such clients. Conversely, a repository that offers no more than a subset of a language's reflection facilities has limited use in that language.
4. *Inconsistency.* Information about a class might be different depending on whether it was obtained from a repository versus reflection. If a program loads a class dynamically, reflection may know about it but not the repository. The repository might provide more accurate information when someone else changes the class.
5. *Inflexibility*. If you have a tool designed to use reflection, can you run it without a compiler? If the tool works with files, can you retarget it to use reflection? How easy is it to divorce class-browsing facilities from the mechanisms for extracting and managing source information? The easier all that is, the more flexible and reusable the tool.

Making the meta-information source pluggable makes it easy to choose one approach and change your mind later. It can also offer the best of both worlds.

3 Application

Clearly, the degree of pluggability is determined by the ease with which a tool can be retargeted to either information source. Conversely, pluggability must cope with changes in the information sources themselves—to a different reflection model, for example, or to a different repository. Figure 2 shows how two tools, each a client of meta-information, can exploit pluggability:

1. *javadoc*[‡], normally a repository client, is retargeted to use reflection.
2. *object-to-component generator* [18], initially dependent on reflection, is retargeted to use a repository.

Pluggability means we can disconnect *javadoc* from its repository source, namely Doclets [24], and plug it into reflection. Similarly, we can disconnect a component generator that was designed to work with reflection and plug it into Doclets.

3.1 A Retargetable *javadoc*

javadoc is a command-line tool that reads a `.java` source file and generates HTML documentation for it. It parses the class declaration and stylized user comments in the source file. By default, *javadoc* generates the documentation according to a

[‡] http://java.sun.com/products/jdk/javadoc

predefined standard template, subject to limited user options. Indeed, early versions gave users little control over its operation. The parse tree was inaccessible, the abstract syntax was not revealed, and altering the tool's processing was not possible.

By JDK 1.2, *javadoc* had undertaken two substantial design changes. First, the internal *javadoc* representation was exposed in a package called *sun.tools.javadoc*. Second, the format of the documentation became customizable. By running *javadoc* with the new command-line option `-doclet` *SomeDocletTool* (where *SomeDocletTool.class* is a user-supplied compiled class), the *SomeDocletTool* program may use the Doclet API to access the internal representation of the parsed file and to control how the parse tree is processed.

For documentation purposes, Sun provides the *javadoc*-generated HTML files for system classes but generally not their source code. If you don't have the source or the HTML files for a certain class, you cannot re-generate the documentation. Even if you have both the source and the HTML files, there is always uncertainty as to whether the version you're using isn't slightly different. Unfortunately, *javadoc* cannot rely on reflection to recover this information despite the functional similarity.

3.1.1 Disparities

The Doclet repository that *javadoc* uses internally resembles reflection in at least two respects:

1. The repository is not persistent: its lifetime is limited to that of the hosting *javadoc*'s execution. The repository lives and dies with each such execution, just as reflective information is available only at run-time.
2. The Doclet API and the Core Reflection API are nearly isomorphic. The Doclet API classes *ClassDoc*, *MemberDoc*, *FieldDoc*, *ConstructorDoc*, and *MethodDoc* correspond to the Core Reflection API classes *Class*, *Member*, *Field*, *Constructor*, and *Method*. The methods correspond closely as well, although they are distributed a little differently in the two class hierarchies.

Although quite similar, the two APIs are incompatible. The JDK design fails to recognize this similarity, let alone providing the "right" abstraction and attempting to unify them. As a result, things tend to be more complex than they need to be, and reuse opportunities are forgone.

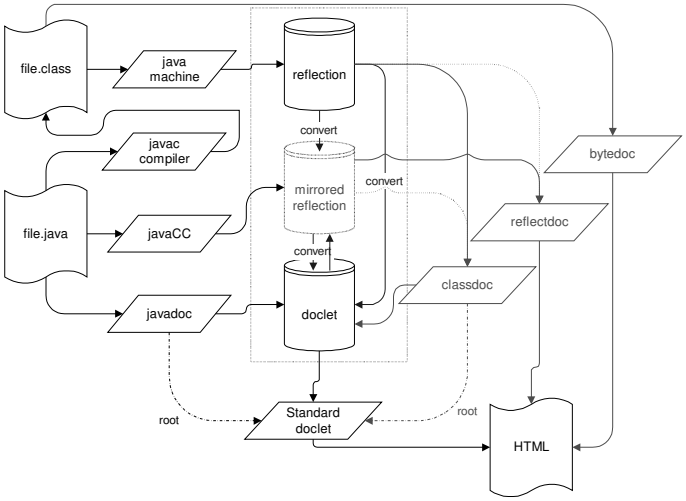

Figure 3: Implementation strategies for a reflective javadoc tool

Here are two examples of difficulties in the design, the first seen from its designer's viewpoint and the second from a client's:

1. The Doclet API's design is newer and more complete than the reflection API's. But since no connection between the two was made, the improvements have not and may never make their way into the reflection API.
2. If you already have *SomeReflectionTool* that adheres to the Core Reflection API, you cannot reuse it with *javadoc*; you must rewrite it as *SomeDocletTool*. The opposite is also true: an existing *SomeDocletTool* cannot be easily applied to reflective information.

   In particular, the *sun.tools.javadoc.doclet.Standard* HTML Doclet, which *javadoc* itself uses by default, cannot be retargeted to `.class` files. Even newly written programs (like your *SomeDocletTool*) cannot use reflection unaltered—a common interface does not exist. There is no way to replace the analyzer. There isn't even a converter in the form of a *ClassDoc* constructor that would translate reflection information into Doclets.

3.1.2 Reconciliation via abstract representation

Figure 4 describes a simple abstract grammar for Java's structural reflection akin to similar representations in Smalltalk [29]. Applying the INTERPRETER pattern [7] to this grammar would yield a precise definition of a class hierarchy similar to `java.lang.reflect`.

Left-hand-side variables are classes; right-hand-side fields are instance variables. In the figure, fields have the general form

Class	::=	*name*:String *superclass*:[**Class**] *interfaces*:**Class**[] **Member**[];
Member	::=	**Field** | **Method** | **Constructor** ;
Field	::=	*declaringClass*:**Class** *name*:**String** *type*:**Class** ;
Constructor	::=	*declaringClass*:**Class** *name*:**String** *ParameterTypes*:**Class**[] ;
Method	::=	*declaringClass*:**Class** *name*:**String** *ReturnType*:**Class** *ParameterTypes*:**Class**[] ;

Figure 4: Grammarized reflection

label:**type**, meaning that the class has an instance variable named *label*. The type of the instance variable can be a primitive type, as in *name*:String, or one of the left-hand-side variables—that is, a reference to another class as in *type*:**Class**.

A type in square brackets is optional, as in *superclass*:[**Class**]. That means an instance of **Class** may have a null *superclass*. Suffixing a type with "[]" describes a container of values. For example, in *interfaces*:**Class**[], *interfaces* is the name of an instance variable of type **Array** of **Class**es. If the label is missing, we assume the instance variable's name to be the plural of the component type. So **Member**[] means there is an instance variable named *members* of type **Array** of **Member**.

The grammar describes a subset of reflection syntax. Java's reflection includes information that is not revealed in this grammar, like **Modifiers** of **Method**s and **Field**s. These were excluded to simplify the exposition.

The grammar displayed in Figure 4 is not, however, an ordinary BNF specification. It is an *abstract* grammar. Information about the concrete syntax of class, fields, and methods is missing. Hence it is worthless for parsing, but we do not intent to parse class declarations with this grammar. (We will use a concrete grammar for that later.) Much of the challenge lies in handling pre-existing instances, such as Java classes.

In sum, this grammar serves as a semantic description of class declarations. An instance of the hierarchy in Figure 4 is better thought of as a typed semantic net than a parse tree. For example, the *superclass* field of **Class** is not an instance variable of type **String** with the name of the superclass. Rather, it is a reference to an instance of type **Class** that is the superclass itself (if one exists). Another example: **Method** knows its ***declaringClass***—information that requires semantic analysis on top of parsing.

3.1.3 Pluggable reflection's benefits

To complete this example, we describe a prototyped version of *javadoc* that generates HTML documentation from reflective information. We demonstrate its usefulness by comparing its output to the official JDK documentation for the Java Core Reflection API. We show how this version of *javadoc,* though designed to work with reflection, also works with a *mirrored* repository with trivial code modifications.[§] As the name suggests, the mirrored repository is fully compatible with the reflection API. We have also built a tool that parses source files and stores them in this repository. We employ these facilities to demonstrate pluggability's advantages for testing and debugging programs that use reflection.

We have two alternative approaches to implementing a reflective version of *javadoc*. The first (re)uses Doclets; the second does not. The reader is encouraged to use Figure 3 as a guide to the examples in particular and more generally to the difficulties conventional reflection creates and the benefits of pluggability.

Now suppose we'd like to make *javadoc* generate HTML documentation using reflection. One tack is to implement a converter from instances of reflection classes to instances of Doclet classes (represented by the downward arrows marked "convert" in the figure). It is possible then to implement a *classdoc* tool that works much like *javadoc* except that it documents classes based on reflective information. Unlike *javadoc*, which works only on `.java` files, *classdoc* may work on `.class` files, hence its name.

A possible implementation strategy has the tool traverse the reflected information, building a temporary Doclet structure, and then passing its root (necessarily of type *sun.tools.javadoc.doclet.Root*) to *Standard.start(Root root)*[**] to generate HTML. Of course, not all Doclet information can be constructed from reflective information. Some entries (e.g., comments) must be left empty or assigned default values. This could make the generated documentation look awkward. To avoid that requires a different implementation approach, one independent of *javadoc* and the Doclet package—a tool that generates the HTML output directly. *reflectdoc* is such a tool. Yet another implementation strategy would be to develop a *bytedoc* for parsing the `.class` files directly without using reflection (see Figure 3).

By applying *javadoc* to files and *reflectdoc* to reflection, you can compare the HTML outputs to reveal unexpected differences. By applying *reflectdoc* to system classes, moreover, you may produce documentation that is not included in the standard distribution. For example, Figure 6 is a portion of the JDK documentation for *java.lang.reflect.Method*. Figure 6 is the HTML documentation generated by running our reflective version of *javadoc*, namely `reflectdoc java.lang.reflect.Method`.

You might appreciate *reflectdoc* just for presenting reflective information in the familiar format of *javadoc*. But note the slight differences. Comments appear in Figure 6 but not in Figure 6. The class modifier *synchronized* appears in Figure 6 but not Figure 6. The same is true of URL links, because *reflectdoc* was run on only `java.lang.reflect.Method`. Future work may include developing "*activedoc*" for producing and controlling *javadoc*'s output dynamically [1].

reflectdoc reveals the following undocumented information about class *Method*:

- Private instance variables: the *Method* class has the variables listed in Figure 7. In the JDK documentation, the *Variables* section does not exist.
- A private constructor: *Method* has the private default constructor shown in Figure 8.
- A static *copy* method whose signature is shown in Figure 9.

Knowledge of private and undocumented features might not be crucial, but it can offer insight into the software's design and a better understanding of reflection. And you never know what you might find until you look. Judging from the API, one might expect class *Class* to have properties like *name*, *superclass*, *interfaces*, and so forth in its implementation. But one might also expect *Class*'s implementation to be lightweight. *reflectdoc*

[§] Only *import* declarations need to be changed.

[**] Oddly enough, the JDK *Standard* class neither inherits from *sun.tools.javadoc.Doclet* nor does it implement any interface that would guarantee the existence of a *start* method with the proper signature.

Class java.lang.reflect.Method

```
java.lang.Object
   |
   +----java.lang.reflect.Method
```

public final class **Method**
extends Object
implements Member A Method provides information about, and access to, a single method on a class or interface. The reflected method may be a class method or an instance method (including an abstract method).

Figure 6: JDK documentation

Class java.lang.reflect.Method

```
java.lang.Object
   |
   +----java.lang.reflect.Method
```

public final synchronized **Method**
extends **Object**
implements **Member**

Figure 6: reflectdoc output

reports that class *Class* has no member variables at all, thus confirming the latter expectation.

Missing information in documentation produced using *javadoc* is often deliberate (e.g., private members are not displayed, nor is the synchronized modifier shown) since these are not part of the interface. But whether certain implementation detail should be hidden is orthogonal to whether you have access to the information.

Variable Index

- **clazz**
- **slot**
- **name**
- **returnType**
- **parameterTypes**
- **exceptionTypes**

Figure 7: Variable Index

Constructor Index

- **Method()**

Figure 8: Constructor index

copy `static Class[] copy(Class[])`

Figure 9: copy method

3.2 Pluggability through Components

We use the object-to-component transformation and automatic generator to produce ReflectionBeans. The input to the generator is Reflection API; ReflectionBeans are the output. In this transformation, every class in the reflection API becomes an event type, and every method becomes a bean. A bean can be a transmitter of an event, a receiver of an event, or both. Most beans both receive and transmit and are hence called *transceivers* [18]. There are several kinds of traversal beans. A bean is considered a *conduit* if it produces only side effects. Two common kinds of conduit beans are *reporter* beans, which generate text, and *predicate* beans, which generate a Boolean event.

If a tool uses ReflectionBeans instead of the reflection interface, then retargeting becomes much easier. Third-party tools can mix and match with third-party providers of meta-information: simply specify the desired `.jar` file in the tool's classpath. This retargeting requires no source code change or recompilation.

ReflectionBeans act as a facade to the actual source, be it reflection, a repository, or anything else. JavaBeans may seem like overkill here, but they have the added benefit of allowing modest visual programming capabilities.

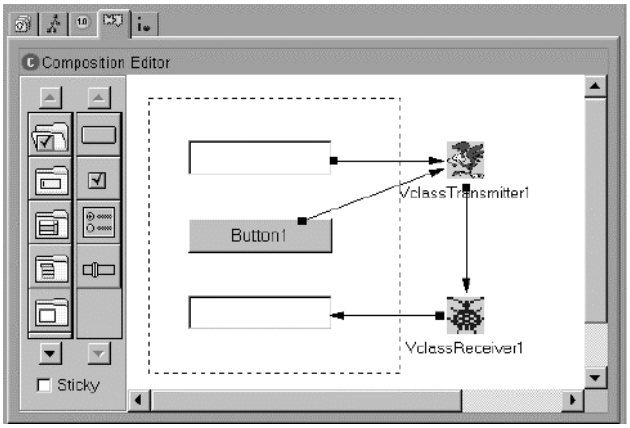

Figure 10: Transmit-Receive

Figure 10 depicts basic cooperation among beans. *VclassTransmitter1* is a *transmitter*-bean [18]: it translates from the AWT-event world to the Visitor-event world. Specifically, it expects a text event and responds with a corresponding class event. The class event encapsulates the class whose name was received in the class event. Internally, it calls on the reflective operation *Class.forName* to do the job. The button *Button1* is connected to the event trigger in *VclassTramsmitter1*.

VclassReceiver1 is a *receiver*-bean: it works like *VclassTransmitter1* but in the opposite direction. On receiving a class event, it fires the name of the class as a text event. Entering a class name in the top text field and pressing the button makes the class name appear in the bottom text field.

This approach is overkill for just propagating a class name, but it illustrates the architecture and its bean-based implementation. A more realistic composition would have nontrivial beans on the communication path between *VclassTransmitter1* and *VclassReceiver1*. Figure 11 shows an example of putting a *Superclass1* in between.

Superclass1 is both a class-event receiver and a class-event transmitter. It accepts and fires class events. It does not fire exactly the same event it receives, however; it is a transceiver-bean as opposed to a *conduit*-bean. Its operation is straightforward. It uses reflection to find the superclass of the incoming class and transmits that superclass as its output. Now if you type a legal class name in the top field and press the button, you would see the name of its superclass in the bottom field.

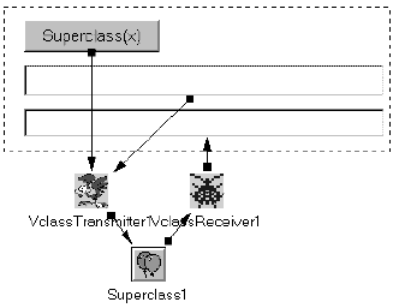

Figure 11: Using the Superclass bean

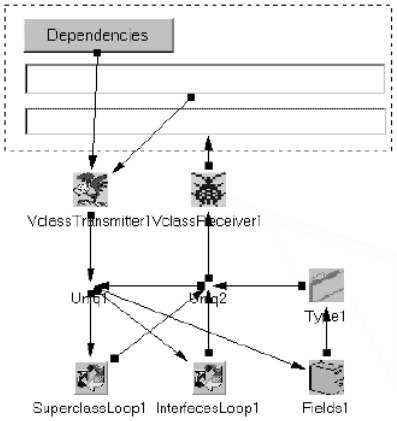

Figure 12: Dependencies

Although individual beans may be simple, they can perform complex jobs in concert. In Figure 12, pressing the button after entering a class name computes the classes it depends on and displays them sequentially in the bottom text field. *SuperclassLoop1* composes a *Superclass* bean and a self-loop. Thus for each input event it outputs multiple events, one for each of the superclasses on the path to the *Object* class. *InterfacesLoop1* does the same for interfaces.

This example generates all the classes that a given class immediately depends on. Given the class *java.lang.reflect.Method*, three class names will flash by: java.lang.Object (*Method*'s superclass), *java.lang.Member* (the interface it implements), and *int* (the type of its instance variable). This example comprises traversal beans exclusively.

When text is entered in the upper field, *TextEvent*s are fired to *VclassTransmitter1*. *VclassTransmitter1* reacts by preparing itself to generate and fire a *ClassEvent* containing a class whose name is the string received.[††] The button labeled "Dependences" is connected to the *fire()* method of *VclassTransmitter1*. When the button is pressed, *VclassTransmitter1* transmits the *ClassEvent* if and only if the string is a legal class name.

Once the button is pressed and *VclassTransmitter1* fires the *ClassEvent*, it is distributed to *Superclass1*, *Interfaces1*, and *Fields1*. *Superclass1* in turn triggers a *ClassEvent* with the superclass of the class. *Interfaces1* triggers multiple *ClassEvent*s, one for each interface the class implements. *Fields1* triggers multiple *FieldEvent*s, one for each public field of the class. The *FieldEvent*s are received by *Type1*, which fires a *ClassEvent* corresponding to the field's type. All the *ClassEvent*s are subsequently merged and sent, one by one, to *VclassReceiver1*. Finally, *VclassReceiver1* generates a *TextEvent*, and the class names are displayed sequentially.

This approach may be applied to *javadoc* to make its source of reflective information pluggable. Figure 14 depicts the construction of *reflectdoc* from JavaBeans. The *reflectdoc* example introduces behavioral beans (counters, resetters, incrementers, and reporters) into the datapath to generate HTML rather than just displaying class names. There are in fact two orthogonal

[††] *VclassTransmitter1* can also be customized directly using a bean customizer.

datapaths here: one governs how reflective information is traversed, and the other governs the flow of data. Our original classification, which assumed a single datapath, differentiated among several types of tranceivers: conduit, converter, transmuter, etc. Here we have two datapaths; hence each bean is characterized by a taxonomic tuple, e.g., <*transmuter, conduit*>. To avoid a Cartesian product of bean classifications, we require that any pair of tranceivers must include a conduit.[‡‡]

These extensions to our taxonomy are needed to characterize some of the beans that implement *javadoc*'s behavior, for example, regarding event synchronization. Further details are beyond the scope of this paper. The important point is that ReflectionBeans are powerful enough to express real applications.

4 Related work

4.1 Mirrors in Self and Smalltalk

The idea of having multiple sources of reflective information is not new. Smith's seminal work [32][33] explicitly accounts for the possibility of differing kinds of causal connection in reflective systems.

The ANSI Smalltalk standard [35] uses an abstract syntax specification to describe the global components of a Smalltalk program and their relationships. In particular, the work on Mirrors in Self [37], Animorphic Smalltalk [36], and Strongtalk [28] (as well as the Mirror interface in JDI[§§]) address concerns similar to ours from a programming language perspective. Mirror objects reflect other objects.

While we use Java as our testbed, these concepts are valid for other languages. Even in a dynamically typed language such as Smalltalk, the implementation and interface are insufficiently decoupled to achieve pluggability. This becomes apparent when trying to retarget reflective Smalltalk code between a 3-metalevel system (e.g., Little Smalltalk) to a 5-metalevel system (e.g., Squeak).

4.2 Intercession

Chiba and Tatsubori [5] extend Java reflection to permit language extensions by re-implementing *java.lang.Class*. They put forth the classes *openjava.mop.OJClass* and *OJMethod* as replacements for *Class* and *Method*. The ability to perform language customization in addition to introspection is achieved using compile-time reflection in their OpenJava compiler [4]. In comparison, we make a point of preserving class names in the mirrored package to ensure that code can run with either the original reflection or the mirrored repository. Instead of creating a dedicated compiler, we can rely on a standard Java compiler. VISITOR furnishes the extended functionality non-invasively.

Javassist [4] supports structural reflection in Java by bytecode manipulation at load-lime without replacing the compiler. Javassist relies on a configuration file and class loading interception. In contrast, we focus on supporting run-time introspection.

4.3 *javadoc* and Demeter

While *javadoc* is widely used, little has been written explaining its usage or design [8][6]. We introduced two variants of *javadoc* (*classdoc* and *reflectdoc*) that use reflection instead of a repository.

More complex tools corroborate the trade-offs and potential benefits of pluggable reflection. Demeter/J [11], for example, is a repository-based tool for adaptive programming (AP). A variant of Demeter/J called *DJ* uses Java's reflection instead of a repository [23] [12]. Its developers report similar trade-offs in using reflection versus a repository. The effort required to implement DJ underscores the duality problem.

Ideally, it would have been trivial to retarget Demeter/J. Unifying the reflection-repository duality can be viewed as an adaptability problem—the raison d'être of the Demeter system. Demeter achieves structure-shy adaptability through *traversal strategies.* This may suggest applying a notion similar to the VISITOR approach to achieve better unification in DJ.

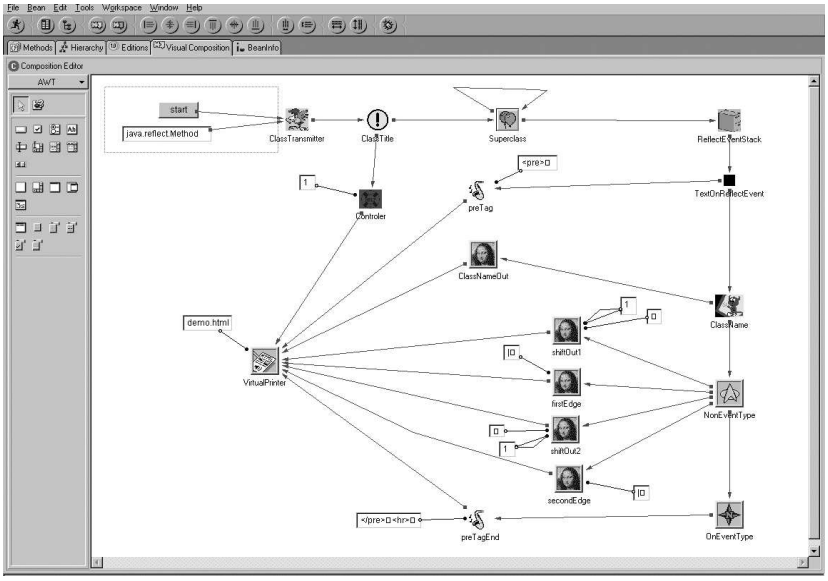

Figure 14: A JavaBeans implementation of javadoc

4.4 AOP and *javadoc*

The componentization problem can be characterized as a separation of concerns—reflective information's use separated from its

[‡‡] More generally, given an *n*-dimensional datapath, all but one dimension must denote a conduit.
[§§] Java Debug Interface

access. In fact, an attempt to apply VISITOR to effect a form of Aspect-Oriented Programming (AOP) [8] was the genesis of this research. We first characterized implicit invocation combined with VISITOR as an aspect-oriented pattern [16]. Then we developed a taxonomy of JavaBeans [18]. This paper builds on those works.

AspectJ, an implementation of AOP for Java, recently provided a *javadoc*-like tool named *ajdoc* [13]. Unlike *javadoc*, *ajdoc* also generates documentation for crosscuts, advice, and introductions. *ajdoc* will also add appropriate aspectual links. *ajdoc* is superior in design to *javadoc*: To deal with the moving target, Palm has (independently) implemented a hidden `-compiler` flag that can be used with *ajdoc* (though it wasn't activated in the public distribution [22].) With a supplemental RootDocMaker and a minor change to *org.aspectj.tools .ajdoc.Ajdoc.java*, one can build the docs from class files. This provides further evidence that pluggable reflection may already be a pattern in use.

Nevertheless, AspectJ code cannot expect to reflect on aspectual structure but must make do with the underlying object structure. Pluggable reflection would be helpful not only for retargeting *javadoc* to reflection but also to let aspects reason about themselves explicitly. This shortcoming is symptomatic of the difficulty in keeping reflection in step with language features. Designing pluggability into the system could avert this problem. Indeed, pluggable reflection can and should become as standard a facility as garbage collection. In the meantime, the implementation we describe here can help bring reflective facilities to new languages that lack them.

4.5 Language Extensions for Components

Jiazzi [20] extend Java with package types as means for implementing units [2]. One could thus use Jiazzi to define a "blueprint" for the Java reflection package. As with AspectJ, the difficulty in using reflection with Jiazzi stems from its being a language extension.

ArchJava [1] addresses the decoupling of implementation code from architecture. New constructs for ports and connections are introduced to enforce, for example, architectural communication-integrity. We aspire to push reflection up to the architectural level, while maintaining "reflective-integrity." Aldrich, et al., mention an *archjavadoc* tool as future work.

4.6 Visual Programming

Everyone who uses visual programming tools experiences their limitations. Building an application with a JavaBeans builder is no different. Once the program gets big, the visual description is no longer manageable. Indeed, the *javadoc* tool is about as large an application as most builders can handle. Nevertheless, visual programming has undeniable appeal for programming in the small—precisely this kind of programming you do with reflection. Instead of learning the textual syntax of reflection in Java, one can describe a portion of an application visually with only the general understanding of the semantics of reflection JavaBeans. However, the visualization is orthogonal to the idea of pluggability.

4.7 Typed Reflection

Reflective code typically uses strings and relies on conventions rather than language mechanisms. Weirich proposes incorporating mechanisms for dynamic type analysis into Java's reflection [27]. In the component approach to pluggability, we assign types to events to achieve typed construction. Typed reflection would make reflective code more concise and safe, but it would not resolve the problem of retargeting a reflective program to another reflective structure or to another language. Unifying the notions of pluggable and typed reflection is a promising track for further research.

5 Conclusion

Reflection in Java can be viewed as the result of taking the abstract grammar of a class definition and applying the INTERPRETER pattern to it, producing a mirrored class hierarchy with the same interface as reflection. We demonstrated several properties of that approach:

1. The mirrored hierarchy can serve as a class repository by generating a parser that converts source code to instances of the mirrored classes.
2. Clients of the reflection interface can be retargeted to use the mirrored hierarchy without modification apart from changing import statements.
3. In particular, Visitors written for the Interpreter pattern are no different than other clients—they can be retargeted to use reflection as well. Reflective programs that use visitors as building blocks thus become pluggable at compile-time.
4. For even looser coupling, reflective programs may use components as their building blocks, affording full run-time pluggability.

The contribution of this paper is more in staging the problem than in solving it. In an evolutionary path from conventional to pluggable reflection, mirroring reflection is merely the first step: a BRIDGE for decoupling users from providers of meta-information. Clients cannot tell the difference between mirrored and genuine reflection. Implicit invocation adds flexibility through looser coupling, allowing third-party composition of reflection information and clients. The Holy Grail, of course, is to give clients uniform access to metadata sources with disparate and unforeseen semantics.

An obvious limitation of mirroring is that the reflection and repository APIs must be kept in sync. Unfortunately, interfaces invariably evolve. A single interface may go through several versions; parallel interfaces may diverge. How do you reconcile two interfaces that are similar but not identical? Programming to the least common denominator may be unsatisfactory because you cannot exploit the unique advantages of either interface. On the other hand, specialized interfaces make switching implementations difficult, thereby forgoing the benefits of encapsulation, portability, and testability.

Both kinds of interface evolution are found in Java's reflection API. Reflection did not exist in JDK 1.0. Had you wanted to implement a class browser, you would have needed a repository.

JDK 1.1 introduced reflection, and you could have migrated your class browser to use it instead. The initial reflection API did not support all class information, however, and so the repository still would have been useful. JDK 1.2 added the ability to access the package that a class belongs to, and it introduced the Doclet API, which can be used instead of a repository. JDK 1.3 added the ability to define proxy classes dynamically, thereby enabling limited behavioral reflection. 1.3 also added the Mirror API, yet another reflective facility as part of the Java Debugging Interface (JDI).

The proliferation of APIs poses a dilemma: How does one design client code that can work with current and future APIs with minimal (ideally zero) change? Printer driver interfaces are a compromise between generic and specialized printing features. Unique or advanced features are typically eschewed to maximize compatibility. As a result, an application that needs a specialized feature must circumvent the printer driver or forgo the feature.

The role of reflection is expanding, and it's more prominent then ever. Components in particular rely heavily on reflection, and clients need to understand how so. With increased awareness of reflection comes the need for supporting tools. Traditional tools for source code, such as code beautifiers, metrics and documentation generators, etc., would be most welcome if they worked on reflection information. Unfortunately, they rarely do. Pluggable reflection can help, as we saw for *javadoc*.

Now imagine a world in which reflection is pluggable. Tools can run on source code files, on class repositories, or on reflection without change and without the information sources' a priori knowledge of each other. Different trade-offs among the sources permit the tool to optimize itself to its context.

Testing would come of age in this world. One scenario: A third-party framework comes with a semantic-checker tool for certifying correct customization of the framework. The checker can run on the source code to catch *syntax* errors. It can also work on class hierarchies to catch *semantic* errors (such as forgetting to subclass a framework class). And it can run on reflection to check for certain semantic violations that (thanks to the halting problem) can only be discovered at run-time. Moreover, the checker can cache and compare test results, as we did with the *javadoc* and *reflectdoc* outputs, to spot inconsistencies. These are some of the benefits we can expect of fully pluggable reflection.

6 Acknowledgment

We thank Gilad Bracha for his comments on an early version of this paper.

7 References

[1] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In *ICSE'02*, May 2002.

[2] M. Flatt and M. Felleisen. Units: Cool modules for hot languages. In PLDI'98, pages 236–248, 1998.

[3] Erik Berglunds. *Dynamic JavaDoc*. http://www.ida.liu.se/~eribe/djavadoc

[4] S. Chiba. Load-time Structural Reflection in Java. In ECOOP 2000, LNCS 1850, Springer Verlag, pages 313–336, 2000.

[5] S. Chiba and M. Tatsubori Chiba. Yet Another `java.lang.Class`. ECOOP '98 Workshop on Reflective Object-Oriented Programming and Systems, Brussels, July 20, 1998.

[6] L. Friendly. The Design of Distributed Hyperlinked Programming Documentation. In Proceedings of the International Workshop on Hypermedia Design (IWHD), 1995.

[7] E. Gamma, et al. *Design Patterns*. Addison–Wesley, Reading, MA, 1995.

[8] G. Kiczales, J. des Rivières, and D. G. Bobrow. The Art of the Metaobject Protocol. MIT Press, 1991.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pages 220–242. Springer Verlag, 1997.

[10] D. Kramer. API documentation from source code comments: a case study of Javadoc. In Proceedings on the seventeenth annual international conference on Computer documentation, Pages 147–153, September 9–14, 1999, New Orleans, LA USA.

[11] K. J. Lieberherr. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company, Boston, 1996.

[12] K. J. Lieberherr and D. Orleans. DJ: Dynamic Adaptive Programming in Java. *In Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 2001.

[13] C. V. Lopes and G. Kiczales. Recent Developments in AspectJ™. In ECOOP'98 Workshop Reader, Springer-Verlag LNCS 1543.

[14] D.H. Lorenz, P. Petkovic: Design-Time Assembly of Run-time Containment Components. TOOLS 34, 2000.

[15] D.H. Lorenz. Tiling design patterns: A case study using the Interpreter pattern. *ACM SIGPLAN Notices* 32(10):206–217, Oct. 1997. In OOPSLA'97, Atlanta, Georgia, Oct. 5–9, 1997.

[16] D.H. Lorenz: Visitor Beans: An Aspect-Oriented Pattern. ECOOP AOP Workshop 1998: 431–432.

[17] D.H. Lorenz, J. Vlissides: Automated Architectural Transformation: Objects to Components. Technical Report NU-CCS-00-01, College of Computer Science, Northeastern University, April 2000.

[18] D.H. Lorenz and J. Vlissides. Designing Components versus Objects: A Transformational Approach. In *Proceedings of the 23rd International Conference on Software Engineer-*

ing, pages 253–262, Toronto, Canada, May 12–19 2001. ICSE 2001, IEEE Computer Society.

[19] P. Maes. Concepts and experiments in computational reflection. ACM SIGPLAN Notices, 22(12):147–155, December 1987. In OOPSLA '87 Conference Proceedings, pages 147–155, 1987.

[20] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New Age Components for Old-Fashioned Java. In OOPSLA 2001. ACM SIGPLAN Notices, Volume 36 , Issue 11, pages 211–222, November 2001

[21] M. E. Nordberg III. Default and extrinsic visitor. In R. Martin, D. Riehle, and F. Buschmann, editors, pages 105–124. Pattern Languages of Program Design 3, Software Patterns. Addison-Wesley, 1998.

[22] J. Palm. *ajdoc*. E-mail message, Mon, 18 Mar 2002 21:51:19 -0700 (MST).

[23] N. Sangal, E. Farrell, K. Lieberherr, and D.H. Lorenz: Interaction Schemata: Compiling Interactions to Code. TOOLS 30, 1999: 268–277.

[24] M. Pollack: Code Generation Using Javadoc: Extend Javadoc By Creating Custom Doclets. JavaWorld, August 2000

[25] M. Shaw and D. Garlan. Software Architecture, Perspectives on an Emerging Discipline. Prentice-Hall, 1996.

[26] J. Visser. Visitor combination and traversal control. In *OOPSLA 2001 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, volume 36, pages 270–282. ACM SIGPLAN Notices, November 2001.

[27] S. Weirich. *Programming With Types*. PhD thesis, Cornell University, 2002. Forthcoming.

[28] L. Bak, G. Bracha, S. Grarup, R. Griesemer, D. Griswold, and U. Holzle. *Mixins in Strongtalk*. In ECOOP 2002 Inheritance Workshop, 2002.

[29] P. J. Caudill and A. Wirfs-Brock. A third generation Smalltalk-80 implementation. In *OOPSLA'86*, pages 119–130. ACM Press, 1986.

[30] J. McAffer. Meta-level programing with CodA. In *ECOOP'95*, pages 190–214, 1995.

[31] M. Richmond and J. Noble. Refletions on remote reflection. In *Proceedings of the 24th Australasian conference on Computer science*, pages 163–170, Gold Coast, Queensland, Australia, 2001. IEEE Computer Society Press.

[32] B. C. Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, Jan. 1982.

[33] B. C. Smith. Reflection and semantics in lisp. In *Proceedings of the 11th ACM SIGPLAN symposium on Principles of programming languages*, pages 23–35, 1984.

[34] A. Wirfs-Brock and B. Wilkerson. An overview of modular Smalltalk. In *OOPSLA'88*, pages 123–134. ACM Press, 1988.

[35] Programming Language Smalltalk Document Number: ANSI/INCITS 319-1998. InterNational Committee for Information Technology Standards (formerlyNCITS), Jan 1, 1998, 292 pages.

[36] G. Bracha and D. Griswold. Extending Smalltalk with mixins, Sept. 1996. OOPSLA Workshop on Extending the Smalltalk Language.

[37] D. Ungar, R. Smith. Self: The power of simplicity. In OOPSLA'87, pages 227-242. Oct. 1987.