Pluggable Reflection: Decoupling Meta-Interface and Implementation

David H. Lorenz*
Northeastern University
College of Computer & Information Science
Boston, MA 02115, USA
lorenz@ccs.neu.edu

John Vlissides
IBM T.J. Watson Research Center
P.O. Box 704,
Yorktown Heights, NY 10598, USA
vlis@us.ibm.com

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 [30, 31, 23] 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. 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 printer 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 third parties, for example, could plug the debugger into either a source code repository or reflection. We call this vision *pluggable reflection*.

The paper demonstrates the potential advantages of pluggability by illustrating 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 [13]). For added flexibility, the second approach implements INTERPRETER dynamically rather than statically using a set of event-based ReflectionBeans components.

Both examples are based on an abstract syntax represen-

*Supported in part by the National Science Foundation (NSF) under Grants No. CCR-0098643 and CCR-0204432, and by the Institute for Complex Scientific Software at Northeastern University.

tation 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 trade-offs between two meta-information sources. In Section 3 we show the benefits of choosing the most appropriate source, and then in Section 4 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 [22] 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.

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. Again, the code that uses reflective information is the same as the code that accesses the reflective representation.

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).

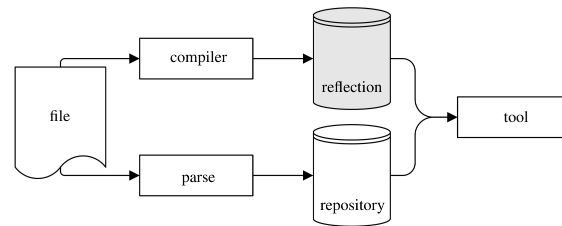

Figure 1. Reflection-repository duality

2.1. Reflection-Repository Trade-offs

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

- *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.
- *Coupling.* Reflection is closely associated with the programming language. As a result, there is 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.
- *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.
- *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.
- *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 infor-

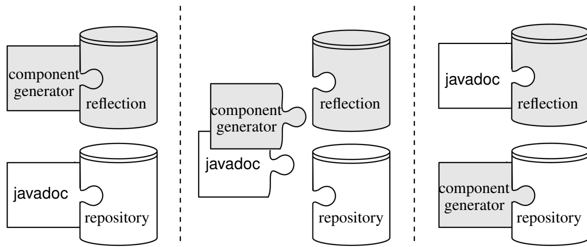

Figure 2. Pluggable reflection

mation 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. Object-to-component generator [22], initially dependent on reflection, is retargeted to use a repository.
2. javadoc [12],[1] normally a repository client, is retargeted to use reflection.

3.1. A Retargetable Component Generator

In previous work, we developed an object-to-component transformation wherein every class in the OO API becomes an event type, and every method becomes a JavaBeans component. 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*. There are several kinds of transceiver beans. A bean is considered a *conduit* if it produces only side effects. A complete classification appears elsewhere [22].

We have also prototyped a component generator that performs this transformation automatically [21]. The generator uses reflection to analyze class interfaces (i.e., relying on method signatures only) and then produces a set of components. Later, we wanted to let clients use the generator as a web service [32]. Unfortunately, Java's reflection lacks interfaces, and retargeting required code modifications. Pluggability would let us disconnect a component generator that was designed to work with reflection and plug it into a metadata repository.

3.2. A Retargetable javadoc

javadoc is a command-line tool that reads a `.java` source file and generates HTML documentation from the class declaration and stylized comments. By default, javadoc generates the documentation according to a predefined standard template, subject to limited user options.

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

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 undergone 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 [28].

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 versions you're using are consistent. Unfortunately, javadoc cannot rely on reflection to recover this information despite the functional similarity.

3.2.1. Disparities

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

- 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.
- 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 pluggable 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. 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.

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

Figure 3. Grammarized reflection

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 front-end analyzer. There isn't even a converter in the form of a `ClassDoc` constructor that would translate reflection information into doclets.

Pluggability would let us disconnect javadoc from its repository source, namely doclets, and plug it into reflection.

4 Pluggability

javadoc can be made pluggable. Section 4.1 illustrates such pluggability by using a unified interface as the abstract representation. An alternative abstract representation using generated ReflectionBeans to implement a pluggable reflective javadoc is illustrated in Section 4.2.

4.1. Reconciliation via Abstract Representation

Figure 3 describes a simple abstract grammar for Java's structural reflection akin to similar representations in Smalltalk [14, 7]. Left-hand-side variables are classes; right-hand-side fields are instance variables. In the figure, fields have the general form *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." 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, such as modifiers of methods and fields. These were excluded to simplify the exposition. The grammar displayed in Figure 3 is not, however, an ordinary BNF specification. It is an abstract grammar. Information about the concrete syntax of classes, fields, and methods is missing. Hence it is worthless for parsing, but we do not intend to parse class declarations with this grammar.

In sum, this grammar serves as a semantic description of class declarations. An instance of the hierarchy in Figure 3 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.

4.1.1. Retargeting legacy reflective code

Applying INTERPRETER to this grammar would yield a precise definition of a class hierarchy, a mirror of `java.lang.reflect`. Legacy code that works with reflection also works with the mirrored repository with trivial code modifications—only import declarations need to be changed. 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.

4.1.2. Writing new retargetable reflective tools

To complete the javadoc example, we describe a prototyped version of javadoc that generates HTML documentation from reflective information, and in Section 4.1.3 we demonstrate its usefulness by comparing its output to the official JDK documentation for the Java Core Reflection API.

We have two alternative approaches to implementing a reflective version of javadoc. The first (re)uses doclets; the

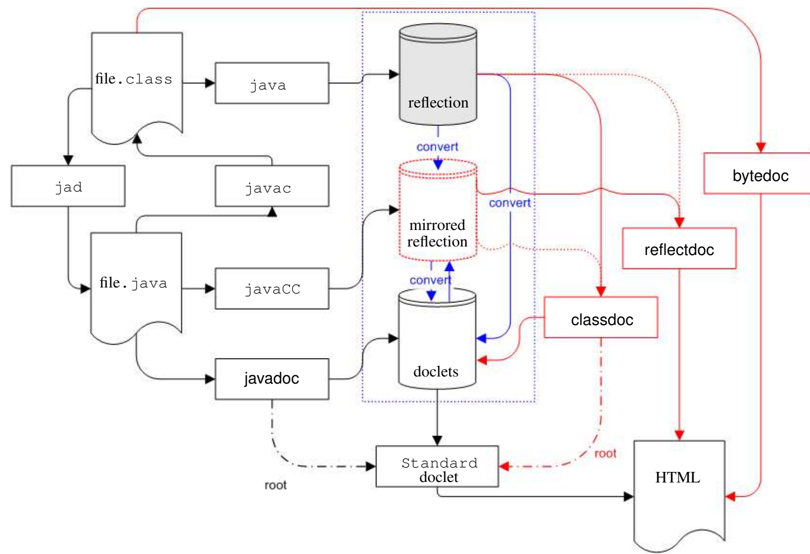

Figure 4. Implementation strategies for a reflective javadoc tool

second does not. The reader is encouraged to use Figure 4 as a guide to the examples in particular and more generally to the difficulties conventional reflection creates and the benefits of pluggability.

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 traversing 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)`[2] 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. Avoiding 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 4).

[2]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.

AspectJ, an implementation of AOP for Java, recently provided a javadoc-like tool named ajdoc [19]. Running `ajdoc` *`SomeAspect`*`.java` generates documentation for the aspect *`SomeAspect`*, which includes aspect instance specifications such as `issingleton`, aspectual links to known advisors and advisees, advice summary, and more.

ajdoc is an application that uses this pluggability approach, and it is superior in design to javadoc. Akin to the `-doclet` option of javadoc, which generates output via an alternate doclet, Palm [27] added a `-compiler` option for generating a `RootDoc` instance via an alternate compiler.[3] Thus the user of ajdoc can control both the front-end and the back-end of the compilation. With a supplemental `RootDocMakerImpl` and a minor change to `org.aspectj.tools.ajdoc.Ajdoc.java`, one can build the doclets from `.class` files. For example, running `ajdoc -standard -compiler RootDocMakerImpl java.lang.reflect.Method` would produce documentation for `Method` via reflection. Nevertheless, AspectJ code cannot expect to reflect on aspectual structure but must make do with the underlying object structure. For example, running `ajdoc -compiler RootDocMakerImpl` *`SomeAspect`* generates documentation using reflection, but the result reveals *`SomeAspect`* to be a class, complete with instrumentation details, instead of displaying the aspectual information.

4.1.3. Pluggable reflection's benefits

By applying javadoc to files and reflectdoc to reflection, you can compare the HTML outputs to reveal un-

[3]The option exists but is unsupported in the AspectJ release.

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 5. JDK documentation for `Method`

Class java.lang.reflect.Method

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

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

Figure 6. `Method` **documentation obtained via** reflectdoc

expected differences. By applying reflectdoc to system classes, moreover, you may produce documentation that is not included in the standard distribution. For example, Figure 5 is a portion of the JDK 1.2 documentation for `java.lang.reflect.Method`. Figure 6 is the HTML documentation generated by running our reflective version of javadoc, specifically `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 5 but not in Figure 6. The class modifier `synchronized` appears[4] in Figure 6 but not Figure 5. The same is true of URL links, because reflectdoc was run only on `java.lang.reflect.Method`. Future work may include developing activedoc for producing and controlling javadoc's output dynamically.

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.

[4]Bug Id 4109635, fixed in JDK 1.3 [33].

Variable Index

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

Figure 7. Variable Index

Constructor Index

- **Method()**

Figure 8. Constructor Index

- 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 you never know what you might find unless 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 reports that class `Class` has no member variables at all,

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

Figure 9. Copy method

thus confirming the latter expectation.

Often, information is deliberately omitted in documentation produced by javadoc (e.g., private members are not displayed), since these are not part of the interface. But whether certain implementation details should be hidden is orthogonal to whether you have access to the information. The `synchronized` modifier bug is an example where even reflective information may be wrong [33].

4.2. Pluggability through Components

The component generator has proven useful for enabling pluggability. So far we've showed pluggability through interface conformity. Now we show an alternative approach to pluggability using components. We demonstrate this again for javadoc by applying the object-to-component transformation and automatic generator to produce ReflectionBeans. The input to the generator is Java's Reflection API; ReflectionBeans are the output.

Using ReflectionBeans instead of the reflection interface makes retargeting 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 depicts basic cooperation among ReflectionBeans. VclassTransmitter1 is a transmitter-bean [22]: it translates from the AWT-event world to the class-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 VclassTransmitter1.

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.

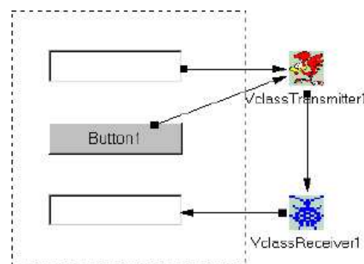

Figure 10. Transmit-Receive

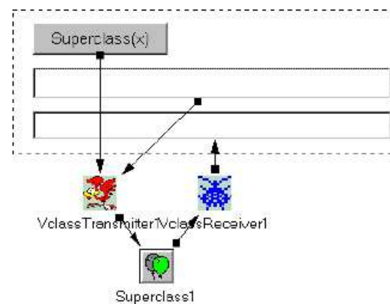

Figure 11. Using the Superclass bean

In Figure 11, a Superclass1 is inserted between those components. 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 may use reflection or any other source of meta-information 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.

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 class' dependents and displays them sequentially in the bottom text field. SuperclassLoop1 composes a Superclass bean and a self-loop. Thus for each input event this composition of beans 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. 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.[5] The button labeled "Dependences" is connected to the `fire()` method of VclassTransmitter1. When

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

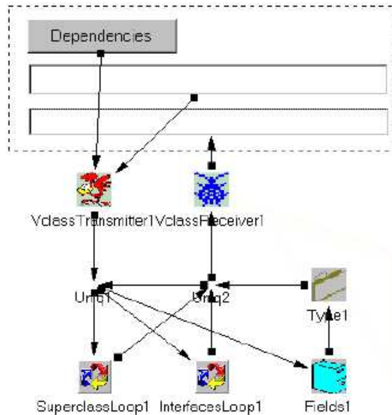

Figure 12. Dependencies

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 *FieldEvents*, 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.

We applied this approach to javadoc to make its source of reflective information pluggable. Implementing reflectdoc using beans introduces behavioral beans (counters, resetters, etc.) into the datapath to generate HTML rather than just displaying class names. There are in fact two orthogonal datapaths: one governs how reflective information is traversed, the other governs the flow of data. Our original classification [22], which assumed a single datapath, differentiated among several types of transceivers: conduit, converter, transmuter, etc. Implementing reflectdoc requires 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 transceivers must include a conduit.[6]

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.

[6]More generally, given an n-dimensional datapath, all but one dimension must denote a conduit.

5. Related Work

Pluggable reflection would be helpful not only for retargeting javadoc to reflection but also to let new language extensions reason about themselves explicitly. This need is symptomatic of the difficulty in keeping reflection in step with language features. Pluggability 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.

5.1. Mirrors in Self and Smalltalk

The idea of having multiple sources of reflective information is not new. Smith's seminal work [30, 31] explicitly accounts for the possibility of differing kinds of causal connection in reflective systems.

The ANSI Smalltalk standard [15] 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 [34], Animorphic Smalltalk [4], and Strongtalk [3] (as well as the `Mirror` interface in JDI) address concerns similar to ours from a programming language perspective [5]. Our extralingual approaches to pluggability should help inform the design and implementation of new reflection mechanisms at the language level.

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 from a 3-metalevel system (e.g., in Little Smalltalk [6]) to a 5-metalevel system (e.g., in Squeak [17]).

5.2. Intercession

Chiba and Tatsubori [9] 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 [8]. 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 [13] furnishes the extended functionality non-invasively.

Javassist [8] 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.

5.3. Aspect-Oriented Software Development

Pluggability can be characterized as a separation of concerns—reflective information's use separated from its access. In fact, an attempt to apply VISITOR to effect a form of Aspect-Oriented Programming (AOP) [16] was the genesis of this research. We first characterized implicit invocation combined with VISITOR as an aspect-oriented pattern [20]. Then we developed a taxonomy of JavaBeans [22]. This paper builds on those works.

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 [18], 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 [29, 26]. Its developers report similar trade-offs in using reflection versus a repository. The effort required to implement DJ underscores the need for pluggability. 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.

5.4. Language Extensions for Components

Jiazzi [24] extends Java with package types as a means for implementing units [11]. 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, 2] 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.

5.5. Visual Programming

Everyone who uses visual programming tools experiences their limitations. Building an application with a JavaBeans builder is no exception. Once the program gets big, the visual description is no longer manageable. Nevertheless, visual programming has undeniable appeal for programming in the small—the 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. The visualization is however orthogonal to the idea of pluggability.

5.6. 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 [35]. 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 tack for further research.

6. Conclusion

People tend to think of reflection as a language feature (and thus coupled to the language) and reflection code as coupled to the language implementation. In this paper, we look at reflection more abstractly as just another interface with the potential for multiple implementations. We consider how to retarget a reflective program to another meta-information source such as a repository, and the benefits of doing so. Indeed, retargetability could even apply across languages: Why couldn't reflectdoc also run on Smalltalk?

To demonstrate reflection as an interface, we took an abstract grammar of a Java class definition and applied the INTERPRETER pattern to it, producing a mirrored class hierarchy with the same interface as reflection. This hierarchy can serve as a class repository by generating a parser that converts source code to instances of the mirrored classes. In particular, visitors written for the INTERPRETER pattern are no different than other clients—they can be retargeted to use reflection as well.

We demonstrated three pluggability approaches:

1. *Legacy clients* of the reflection interface can be retargeted to use the mirrored hierarchy without modification apart from changing import statements.

2. *New clients* that use visitors as building blocks are pluggable at compile-time.

3. For even looser coupling, *third-party clients* may use components as their building blocks, affording full assembly-time pluggability.

The key contribution of this paper lies in staging the problem. In an evolutionary path from conventional to pluggable reflection, mirroring reflection is merely the first step:

a BRIDGE [13] 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 meta-data 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. JDK 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.

Meanwhile, the role of reflection is expanding. 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, and the like would be most welcome if they worked on reflection. Unfortunately, they rarely do.

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-checking 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.

Acknowledgment

We thank Gilad Bracha and the anonymous referees for their valuable comments. Many thanks to Gennady Agranov, Gilad Bracha, Sergei Kojarski, and Jeffrey Palm for helpful discussions on reflection. The reflectdoc tool, the JavaBeans-based retargetable javadoc, and the reflection-based component generator were initially implemented by Predrag Petkovic, and later re-implemented by students in COM3240. Ramanathan Sundaram retargeted the component generator from using reflection to work with meta-information as a web service.

References

[1] J. Aldrich, C. Chambers, and D. Notkin. Component-oriented programming in ArchJava. In D. H. Lorenz and V. C. Sreedhar, editors, *Proceedings of the First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, pages 1–8, Tampa Bay, Florida, Oct. 15 2001. Technical Report NU-CCS-01-06, College of Computer Science, Northeastern University, Boston, MA 02115.

[2] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, Orlando, Florida, May 19-25 2002. ICSE 2002, ACM Press.

[3] L. Bak, G. Bracha, S. Grarup, R. Griesemer, D. Griswold, and U. Hölzle. Mixins in Strongtalk. In A. P. Black, E. Ernst, P. Grogono, and M. Sakkinen, editors, *Proceedings of the Inheritance Workshop at ECOOP 2002*, Málaga, Spain, June 11 2002. ECOOP 2002, University of Jyväsjylä Information Technology Research Institute.

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

[5] G. Bracha and D. Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. Unpublished manuscript, 2003.

[6] T. Budd. *A Little Smalltalk*. Addison-Wesley, 1987.

[7] P. J. Caudill and A. Wirfs-Brock. A third generation Smalltalk-80 implementation. In *Proceedings of the 1st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 119–130, Portland, Oregon, USA, Sept. 29 - Oct. 2 1986. OOPSLA'86, ACM SIGPLAN Notices 21(11) Nov. 1986.

[8] S. Chiba. Load-time structural reflection in Java. In E. Bertino, editor, *Proceedings of the 14th European Conference on Object-Oriented Programming*, number 1850 in Lecture Notes in Computer Science, pages 313–336, Cannes, France, June 12-16 2000. ECOOP 2000, Springer Verlag.

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

[10] S. Demeyer and J. Bosch, editors. *Object-Oriented Technology. ECOOP'98 Workshop Reader*, number 1543 in Lecture Notes in Computer Science. Workshop Proceedings, Brussels, Belguim, Springer Verlag, July 20-24 1998.

[11] M. Flatt and M. Felleisen. Units: Cool modules for hot languages. In *Proceedings of the conference on programming language design and implementation*, pages 236–248, Montreal, Quebec, Canada, May 1998. PLDI'98, ACM Press.

[12] L. Friendly. The design of distributed hyperlinked programming documentation. In *Proceedings of the International Workshop on Hypermedia Design (IWHD)*, 1995.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing. Addison-Wesley, 1995.

[14] A. Goldberg and D. Robson. *Smalltalk–80: The Language*. Addison-Wesley, Reading, MA, USA, 1989.

[15] InterNational Committee for Information Technology Standards (formerly NCITS). *Programming Language Smalltalk*, Jan. 1 1998.

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, June 9–13 1997. ECOOP'97, Springer Verlag.

[17] G. Korienek, T. Wrensch, and D. Dechow. *Squeak - A Quick Trip to ObjectLand*. Addison–Wesley Publishing Company, 2001.

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

[19] C. V. Lopes and G. Kiczales. Recent developments in AspectJ. In Demeyer and Bosch [10], pages 398–401.

[20] D. H. Lorenz. Visitor Beans: An aspect-oriented pattern. In Demeyer and Bosch [10], pages 431–432.

[21] D. H. Lorenz and J. Vlissides. Automated architectural transformation: Objects to components. Technical Report NU-CCS-00-01, College of Computer Science, Northeastern University, Boston, MA 02115, Apr. 2000. `http://www.ccs.neu.edu/home/-lorenz/papers/reports/NU-CCS-00-01.html`.

[22] D. H. Lorenz and J. Vlissides. Designing components versus objects: A transformational approach. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 253–262, Toronto, Canada, May 12-19 2001. ICSE 2001, IEEE Computer Society.

[23] P. Maes. Concepts and experiments in computational reflection. In OOPSLA'87 [25], pages 147–155.

[24] S. McDirmid, M. Flatt, and W. C. H. Jiazzi. New age components for old-fashioned java. In *Proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 211–222, Tampa Bay, Florida, Oct. 14-18 2001. OOPSLA'01, ACM SIGPLAN Notices 36(11) Nov. 2001.

[25] OOPSLA'87. *Proceedings of the 2nd Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Orlando, Florida, Oct. 4-8 1987. ACM SIGPLAN Notices 22(12) Dec. 1987.

[26] D. Orleans and K. J. Lieberherr. DJ: Dynamic adaptive programming in Java. In A. Yonezawa and S. Matsuoka, editors, *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Reflection 2001*, number 2192 in Lecture Notes in Computer Science, pages 73–80, Kyoto, Japan, Sept. 25-28 2001. Springer Verlag.

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

[28] M. Pollack. Code generation using javadoc: Extend javadoc by creating custom doclets. *JavaWorld*, Aug. 2000. `http://www.javaworld.com/javaworld/jw-08-2000/jw-0818-javadoc.html`.

[29] N. Sangal, E. Farrell, K. Lieberherr, and D. H. Lorenz. Interaction schemata: Compiling interactions to code. In *Proceedings of the 30th International Conference on Technology of Object-Oriented Languages and Systems*, pages 268–177, Santa Barbara, CA, Aug. 1-5 2000. TOOLS 30 USA Conference, IEEE Computer Society.

[30] B. C. Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, MIT LCS TR-272, Jan. 1982.

[31] 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.

[32] R. Sundaram and D. H. Lorenz. Meta-data driven code generator web service. In P. Tarr, A. Finkelstein, B. Hailpern, G. Piccinelli, and J. Stafford, editors, *Proceedings of the OOPSLA 2002 Workshop on Object-Oriented Web Services*, Seattle, Washington, Nov. 5 2002.

[33] A look at the synchronized modifier. *JavaWorld*, June 1999. `http://www.javaworld.com/javaworld/javaqa/1999-06/03-synchronized.html`.

[34] D. Ungar and R. B. Smith. Self: The power of simplicity. In OOPSLA'87 [25], pages 227–242.

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