Reflection in Java, CORBA und JacORB

Gerald Brose

Freie Universität Berlin, Institut für Informatik, Takustraße 9,
D–14195 Berlin, Germany
brose@inf.fu-berlin.de

Abstract. Reflection has emerged as a very flexible programming technique and a structured way of achieving program adaptability. In this paper we describe similarities and differences between the reflective facilities available in Java and CORBA and present an implementation of the CORBA Interface Repository for JacORB which makes use of Java reflection.

1 Introduction

Many modern applications, especially in distributed settings, have to be designed in an *open* way such that they can flexibly and dynamically accomodate to changes in requirements, configuration or in their environment. A promising technology for achieving this necessary kind of flexibility and adaptability in a structured way is *reflection*. Many modern programming languages and platforms come with reflective facilities, and so does Java.

As CORBA [11] has established itself as an important standard for distributed object oriented systems, it is also interesting to look at what reflective facilities are prescribed in a CORBA implementation. This paper gives a short overview of reflection in general and describes the role of reflection in both CORBA and Java. It also examines the use of the Java reflection API in the design of a CORBA Interface Repository component for the Java–CORBA implementation JacORB[3].

The remainder of this paper is organized as follows: Section 2 gives a short introduction to reflection and the terminology used. Section 3 examines the role of reflection in the CORBA architecture. In section 4 we describe the possibilities for reflection in Java and compare them to CORBA. Finally, we introduce JacORB and describe our design for a CORBA Interface Repository in section 5. The paper concludes with a summary.

2 Reflection

The terms *reflection* or *meta–level programming* are generally used for systems that have the ability to reason about themselves, using some kind of self–representation. This reasoning is done at a *meta level* where certain aspects of the system are represented or *reified* as *meta objects*. The code dealing with

meta objects is called a *meta–program* and the interface to the meta objects is called the *metaobject protocol* [7].

On the one hand, separating meta–level code from base–level code can help separating concerns. Non–functional system aspects such as, e.g., security or persistence can be addressed in the meta–program, thus enhancing base–level code reusability. On the other hand, meta–level concepts can provide for more flexibility and better adaptability to changing environments. One prominent application area for reflection are programming tools, e.g. browsers, debuggers or prototyping environments.

Usually, the term reflection is understood to mean *computational reflection* [9], i.e. meta objects allow to reflect about the computational process by giving first–class status to concepts that are implicit in the programming model, e.g. by reifying classes, interfaces or method invocations.

Implementational reflection [13] allows a program to access aspects of the system implementation. Here, meta objects are specifically tailored to represent system–level structures and allow programs to inspect or even change aspects of their own implementation, e.g. by modifying garbage collection strategies or thread scheduling. While certain system aspects are hidden from most applications underneath a carefully designed abstraction layer, those programs that do require access to lower–level features are given a well–defined way to use them. The code using only the abstractions provided by the system can then be defined as belonging to the base level while those parts that use the lower–level interfaces constitute the meta program.

For computational reflection, meta objects can further be categorized using the terminology of [5] which describes the relation between base–level and meta–level objects. In the *meta–class model* as, e.g., represented by Smalltalk, a meta object is attached to the class of an object. As a consequence, all instances of a class share the same meta object. The *specific meta–object model* distinguishes between meta objects and classes and thus allows individual objects to have their own meta objects. In the *meta–communication* model, not objects but method invocations are reified.

Reflective facilities may range from mere *inspection* of selected system properties to very powerful and flexible means of *adapting* or extending a system or programming model. In [5], these different reflective approaches are termed *structural* and *computational reflection*. To avoid confusion with the more general term, the latter is also called behavioral reflection. Whereas *structural reflection* allows to inspect the structural dimensions of a system — such as classes, inheritance and instantiation relationships in an OO–system — *behavioral reflection* is concerned with observing or changing the behavior of a system, e.g. by redirecting method invocations, adding new methods to classes at run time or by allowing objects to dynamically change their type.

Another criterion for classifying reflectional systems is how and when reflection is actually triggered. This aspect is expressed through the distinction between *explicit* vs. *implicit reflection* [10] which describes whether switching to meta–level computations is done by explicitly initiating reflective computation,

e.g. by particular API calls. In contrast, implicit reflection means that the system will reflect at certain predefined stages during processing, e.g. whenever a method is invoked.

A more general distinction is between compile–time and run–time reflection. In compile–time reflection, the meta program is given access to meta objects that describe the static program structure, e.g. the compiler's abstract syntax tree. Depending on the particular metaobject protocol, the meta program can now inspect this structure as, e.g., in [1], or change it as, e.g., in *Aspect–Oriented Programming* [8], where code is inserted for system aspects that should be transparent to programmers. While run–time reflection is more flexible because it allows access to individual object instances which is not possible with compile–time reflection, it does of course have implications for performance.

3 Reflection in CORBA

Since CORBA is basically a language independent specification of an abstract object model, reflective facilities in CORBA are defined entirely in terms of the CORBA object model and independent of any particular programming environment and its facilities. We will only be concerned with run–time reflection here because the process of compiling IDL specifications and its intermediate data structures are not standardized, so there is no standardized model of compile–time reflection.

CORBA itself is not a reflective architecture in the sense that it would allow to extend its object model or to modify the entire system behaviour in arbitrary ways. In fact, the term reflection is not used anywhere in the specification, and CORBA does not specify a general model of defining arbitrary meta objects for individual instances as in the *specific meta–object model*.

CORBA does, however, heavily rely on meta information in a number of areas, most prominently for run time type information, and thus exhibits aspects of the *meta–class model*. Additionally, its interceptor concept can be classified as a *meta–communication model*. We will look at each of these models in turn. While we will not focus on implementational reflection here, it should be noted that CORBA implementations generally also provide interfaces to access system level aspects, e.g. for setting a threading policy for multi–threaded servers.

3.1 Meta Communication

Remote object invocations are implemented in terms of a request–reply message passing protocol. CORBA standardizes a number of message formats in its GIOP protocol and defines a transfer syntax (CDR). To make reified invocations accessible, CORBA defines a concept called *interceptors*.[1]

[1] Before interceptors were standardized, this concept was offered as a proprietary extension in different ORBs and was called *filters* or *transformers*.

Interceptors are an *implicit* reflectional concept as they are invoked automatically by the ORB at different stages during method invocations. Meta–programmers can write their own interceptors and register these with the ORB so that they will be called at different stages on the client as well as the server side. When invoked, an interceptor is given access to the request or reply meta object and can either inspect, alter or even redirect it to a different object. The most important use for interceptors is for cryptographic message protection and for access control, but in principle they can be used for a wide range of tasks, e.g. profiling, tracing, auditing/logging or redirecting messages. Thus, they constitute a powerful concept for computational reflection and enable full behavioral reflection.

3.2 Meta Classes

As CORBA has no class concept but only object types or interfaces, it might be more appropriate to talk about *meta interfaces* rather than meta classes, but essentially this corresponds to the meta–class model. Every CORBA object supports the operation `get_interface()` which returns a meta object describing the object's type. The `InterfaceDef` object returned by the `get_interface` operation allows to inspect and — at least theoretically — modify the entire type information dynamically. All objects of a certain type in a domain will share the same meta object describing their interface. Because `InterfaceDef` objects cannot describe implementational or behavioral aspects of a type, this kind of reflection is structural rather than behavioral.

Structural reflection — accessing run–time type information through meta objects — is essential in CORBA. One reason is that for interoperating ORB domains with independently administered type systems, there must be a way of determining type equivalence or conformance not only on the basis of type names, but on the basis of structural type properties. Another reason is that it must be possible to build applications that are independent of static type information. This is a requirement for long–lived applications that need to be adaptable, or for generic applications like browsers or application–level bridges or gateways. Such applications need to have a way of invoking objects the types of which were unknown at compile time. To offer this kind of functionality, the *Dynamic Invocation Interface* (DII) component needs access to run–time type information.

Run–time type information in CORBA is managed by the ORB's *Interface Repository* (IR) component. It allows to request, inspect and modify type information. The IR is a separate CORBA object that is remotely accessible and offers operations to retrieve and modify type information. In every ORB domain, there must be at least one such repository. The `get_interface` operation on object references that was mentioned above actually returns an object from the IR. The IR will be described in more detail in the following section.

Type Information in the IR The IR manages type information in a hierarchical containment structure that corresponds to the structure of scoping

constructs in IDL specifications: modules contain definitions of interfaces, structures, constants etc. Interfaces in turn contain definitions of exceptions, operations, attributes and constants. Figure 1 illustrates this hierarchy.

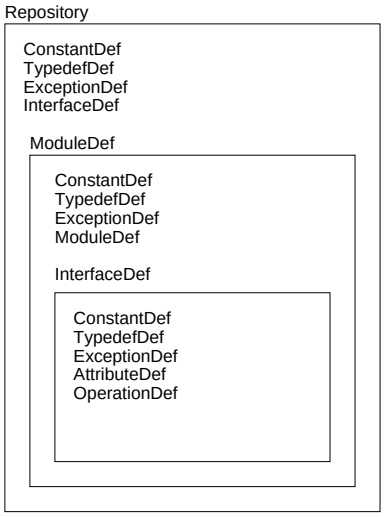

Fig. 1. Containers in the Interface Repository

The descriptions inside the IR can be identified in different ways. Every element of the repository has a unique, qualified name which corresponds to the structure of name scopes in the IDL specification. An interface `I1` which was declared inside module `M2` which in turn was declared inside module `M1` thus has a qualified name `M1::M2::I1`. The IR also provides another, much more flexible way of naming IDL constructs using *RepositoryIds*. There are a number of different formats for RepositoryIds but every Repository must be able to handle the following format, which is marked by the prefix `"IDL:"` and also carries a suffix with a version number, as in, e.g., "`IDL:jacorb/demo/grid:1.0`". The name component between the colons can be set freely using the IDL compiler directives `#pragma prefix` and `#pragma ID`. If no such directive is used, it corresponds to the qualified name as above.

Using the IR When an object's client calls the `get_interface()` operation, the ORB consults the IR and returns an `InterfaceDef` object that describes the object's interface. Using `InterfaceDef` operations on this description object, further description objects can be obtained, such as descriptions for operations or attributes of the interface under consideration.

The IR can also be called like any other CORBA object and provides `lookup()` or `lookup_name()` operations to clients so that definitions can be searched for,

given a qualified name. Moreover, the complete contents of individual containers (modules or interfaces) can be listed.

Interface Repository meta objects provide further description operations. For a given `InterfaceDef` object, we can inspect the different meta objects contained in this object (e.g., `OperationDef` objects). It is also possible to obtain descriptions in form of a simple structure of type `InterfaceDescription` or `FullInterfaceDescription`. Since structures are passed by value and a `FullInterfaceDescription` fully provides all contained descriptions, no further —possibly remote — invocations are necessary for searching the structure.

4 Reflection in Java

Java 1.1 offers a dedicated API for accessing meta objects. Basically, this API represents a meta–class model and allows explicit structural reflection in a similar way as the CORBA Interface Repository while not achieving its full functionality. This section gives a short summary of structural reflection in Java and also presents two non–standard approaches that allow implicit and behavioral reflection.

4.1 Structural Reflection in Java

The Java reflection API consists of the classes in the package `java.lang.reflect` and the class `java.lang.Class`. Using this API, it is possible to retrieve run time type information. All Java objects are instances of the class `Object` which provides the operation `getClass()` to obtain a meta object of class `Class` that represents the object's class. Similar to the operations of the CORBA `InterfaceDef` interface, `Class` objects provide operations to retrieve names, types of attributes and methods. Also, as in CORBA, it is possible to retrieve a `Class` object using its qualified name using the operation `Class.forName()`.

Having the reflection API reside in the `java.lang` package suggests that reflection is an integral part of the Java language, but this is not true to the extent that it is for CORBA. Reflection was added to Java rather late in the language design process, and Java does not actually depend on it. Java itself is not a distributed language, and RMI is not a platform for heterogeneous distribution. Therefore, no language independent type representation is necessary. Also, because RMI allows to download classes dynamically over the network [14], no separate, remotely accessible IR component is needed.

While a distributed type system is not an issue in Java, flexible adaptability of applications and dynamic loading and binding of code is. Consequently, the Java reflection API offers capabilities similar to the CORBA DII to allow method invocations to be constructed at run time by operating on meta objects. Also, attribute values can be set and retrieved and new class instances can be created by calling constructors. For obvious reasons, this is not possible in CORBA: Because a given interface representation is not connected to any one implementation, no new objects can be created from it.

The main difference to the CORBA IR is that constructs cannot be named as flexibly. In Java, the names of all constructs are determined by the static structure of class and package names. There is no way to refer to method or attribute definitions using a globally unique identifier and there is no support for class versioning.[2] Another important difference between Java reflection and the CORBA IR is that Java meta objects cannot be modified. There is, e.g., no way to move an interface from one package to another or to dynamically add an attribute to an interface.[3]

4.2 Other Approaches

As has been pointed out, Java reflection does only offer explicit, structural reflection interfaces. MetaJava [6] provides an extended Virtual Machine implementation that extends Java with facilities for implicit and behavioral reflection. Meta objects can be explicitly associated with objects and classes and are registered with certain base level *events* that represent standard language mechanisms such as method invocations, variable access, object locking or unlocking, class loading or instance creation. When an event occurs for which a registered meta object exists, this meta object's code for the particular mechanism is executed in place of the standard mechanism. This approach is both more general and more powerful than CORBA's interceptor model. Interceptors are part of a metacommunication model, i.e. reflection only takes place at certain points during operation invocation while MetaJava offers more "entry points" to the meta–level. Also, interceptors only allow to *add* behaviour, whereas in MetaJava it is possible to replace an entire mechanism by one of your own devising.

MetaJava, as well as the reflective facilities in standard Java or CORBA, are run time reflectional systems. Barat [1], in contrast, is an approach for compile–time reflection. In this system, the meta program is given access to static program constructs, such as classes, methods, attributes, expressions, variable accesses or method invocations. In principle, the meta objects are the nodes of the program's abstract syntax tree which the meta program may traverse, analyse or reorganize. Barat is specifically geared towards checking additional constraints at compile time that could not be expressed using the standard Java type system, e.g. design constraints such as "all instance variables must be declared private, public access is through accessor methods only".

5 Reflection in JacORB: The IR

JacORB is a freely available[4] implementation of the CORBA standard and is written entirely in Java. Because it contains no native code, JacORB runs without modification on every platform that offers a Java Virtual Machine. JacORB

[2] Java 1.2 offers support for package versioning, but not for individual elements in a package.

[3] This possibility does, in principle, exist in CORBA, but seems not to be implemented because of the obvious consistency problems.

[4] `http://www.inf.fu-berlin.de/~brose/jacorb`

implements an ORB with extensive concurrency support, a BOA, DII, DSI, IR and an IDL compiler and stub generator. By using IIOP as its native communication protocol, JacORB interoperates directly with other ORB implementations. Additionally, an OMG–compliant name service and a prototype of an event service are provided.

JacORB originated in an early class library which was designed to provide Java programs with facilities for simple remote method invocations. When it was extended to provide CORBA concepts, the basic technique of generating stub classes from Java class files rather than from IDL interfaces was preserved. To generate source code for stub classes the JacORB stub generator parses Java byte code which was compiled from classes that were either generated with the IDL compiler or manually written. As a consequence, JacORB can support distributed Java programs on top of an IIOP run–time system without requiring a separate interface definition language. This way of writing distributed programs in Java is similar to RMI. However, RMI does not currently support IIOP, while JacORB, on the other hand, does not have RMI's object serialization facilities.

For a full–fledged CORBA system, however, it is not sufficient to be able to remotely access Java objects using IIOP. To provide full CORBA functionality, an Interface Repository had to be built that implements CORBA's structural reflection features. Interceptors are also implemented in JacORB, but we will focus on the design of the Interface Repository in the rest of this section.

Repository Design When designing the Interface Repository, our goal was to exploit the Java reflection API's functionality to avoid having to implement an additional data base for IDL type descriptions. As it turned out, this was possible because the similiarities between the Java and CORBA object models allow to derive the required IDL information at run time. As a consequence, we can even do without any IDL at compile time. However, this required a few modifications to the IDL–to–Java language mapping so that our own language mapping does not fully comply to the standard defined in [11].

In addition to this simplification, the main advantage of our approach lies in avoiding redundant data and possible inconsistencies between persistent IDL descriptions and their Java representations, because Java classes have to be generated and stored anyway.

Thus, the Repository has to load Java classes, interpret them using reflection and translate them into the appropriate IDL meta information. To this end, the repository realizes a reverse mapping from Java to IDL. Figure 2 illustrates this functionality, where f^{-1} denotes the reverse mapping, or the inverse of the language mapping. This mapping is summarized in the following section.

Reverse Mapping Upon receiving a request for an IDL definition, the IR first tries to find the Java class that represents this IDL construct or contains its representation. The Java class name is derived from the RepositoryId or the qualified name given in the request. This class is then loaded if not already present.

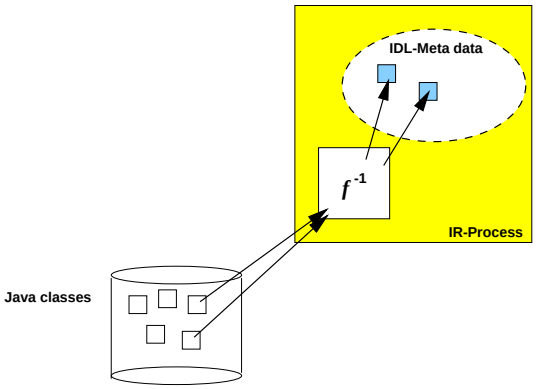

Fig. 2. The function of the JacORB Interface Repository

When loading modules and interfaces that in turn contain other constructs, the generated meta objects have to be created and initialized in two phases to be able to resolve cyclic dependencies between IDL constructs. First, all meta objects must be created along the containment hierarchies, but without resolving references to other repository elements. This has to be done in a separate second phase so that all required meta objects really exist.

During this loading process, it is the IR's task to inspect loaded classes and determine which kind of IDL construct they represent. It then has to create the appropriate meta objects. In principle, this is also possible with Java classes that were generated with the standardized Java–to–IDL mapping. There are, however, a few places where the generated code would not contain enough information to allow a straightforward reverse mapping.

For this reason, the JacORB IDL compiler employs a slightly modified Java–to–IDL mapping to insert that information into appropriate places in the generated code. In order to be able to tell whether a method in a Java interface corresponds to a regular or a `oneway` operation of an IDL interface, the IDL compiler inserts a marker attribute into the generated code. As a further simplification of the reverse mapping, compiler generated classes contain explicit markers that indicate which kind of IDL construct a given class represents. Classes that were generated from IDL `structs`, e.g., implement the interface `jacorb.Orb.Struct`.

As a side effect of our approach, IDL information can also be derived for Java objects that are not instances of IDL compiler generated classes but were separately developed, in particular plain Java interfaces. It can thus also assist in distributing existing Java code. In these cases, those IDL constructs for which no Java equivalent exists will never be derived, unless Java developers insert the necessary markers into their code intentionally. To write a Java class such that the reverse mapping will derive IDL information for a union, e.g., developers would have to declare that class as implementing `jacorb.Orb.Union` and comply to the canonic class format for unions as prescribed by the Java–to–IDL mapping, such as the way the discriminator is modelled in Java, etc.

A somewhat different and more general approach to reverse mapping Java to IDL is taken in [12]. This proposal is intended to serve as a bridge between RMI and CORBA and aims at making Java/RMI objects accessible as regular CORBA objects, taking manually written Java/RMI code as input. In contrast, our mapping is aimed mainly at IDL compiler generated Java classes.

6 Summary and Future Work

We provided an overview of the reflective facilities of CORBA and Java and described a Java implementation of a CORBA Interface Repository that does not need to manage its own persistent storage, thus avoiding redundancies and requiring less resources. The possibility of dynamically deriving IDL from Java classes by using Java reflection proves both the conceptual similarity between Java and CORBA and the practical usefulness of the reflection library. An interesting by–product is the option of distributing existing Java code with CORBA, although this has only been tested with toy examples yet.

With the implementation described above it is not currently possible to construct RepositoryIds that do not correspond exactly to the qualified name of an IDL definition. Because of this limitation the JacORB IDL compiler does not presently support the IDL pragmas `prefix` and `ID`. We believe that this problem can be solved in a future version by modifications to the class loading process. Another restriction is that our repository implementation does not allow to modify the contents of the repository dynamically.

The implementation of the JacORB Interface Repository was complemented by student work on implementing a browser for graphically displaying the IR contents. This tool is implemented in Java, looks like a familiar class browser and facilitates searching the repository contents. An interesting perspective would be to build a development environment for CORBA that integrates IDL compiler, editor, browsers and code generators for CORBA servers.

We plan to further enhance JacORB with an implementation of the *Portable Object Adapter* (POA) and a language mapping that makes use of new language features in Java 1.1 and 1.2. The standardized Java language mapping is based on Java 1.0.2 and thus cannot exploit nested classes and package meta objects. Our main interest for future research with JacORB is in exploring security in CORBA systems.

References

1. Bokowski, B.: Barat — A Front–End for Java. Technical Report B–98–09, Freie Universität Berlin, September 1998
2. Brose, G., Bokowski, B.: Ein Object Request Broker für Java. Informatik/Informatique, Zeitschrift d. schweiz. Informatikorganisationen, **3** (1997), 27–30
3. Brose, G.: JacORB — Design and Implementation of a Java ORB. Proc. Distributed Applications and Interoperable Systems DAIS'97, Cottbus, Germany, September 1997. Chapman & Hall, 143–154

4. Brose, G.: Java & CORBA — How close are they really? Java Developer's Journal, **3** (1998), 60–62
5. Ferber, J.: Computational Reflection in Class based Object Oriented Languages. Proc. OOPSLA 1989, SIGPLAN Notices, ACM Press, 1989, 317–326
6. Golm, M.: Design and Implementation of a Meta Architecture for Java. Master's Thesis, Universität Erlangen–Nürnberg, January 1997.
7. Kiczales, G., de Rivières, J., Bobrow, D.: The Art of the Metaobject Protocol. MIT Press, 1991
8. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingiter, J.-M., Irwin, J.: Aspect–Oriented Programming. Proc. ECOOP 1997, Springer LNCS, June 1997
9. Maes, P.: Concepts and experiments in computational reflection. Proc. OOPSLA 1987, 22, SIGPLAN Notices, ACM Press, 1987, 147–155
10. Maes, P.: Issues in computational reflection. in: Maes, P., Nardi, D. (eds.): Meta–Level Architectures and Reflection. Elsevier 1988, 21–35
11. OMG: The Common Object Request Broker: Architecture and Specification. revision 2.2, February 1998
12. OMG: Java to IDL Mapping. Joint Submission, January 1998
13. Rao, R.: Implementational Reflection in Silica. Proc. ECOOP 1991, LNCS, Springer, Berlin, 1991, 251–267
14. Sun Microsystems: Java Remote Method Invocation Specification. October 1997.