metaXa and the Future of Reflection

Michael Golm, Jürgen Kleinöder
University of Erlangen-Nürnberg
Dept. of Computer Science 4 (Operating Systems)
Martensstr. 1, D-91058 Erlangen, Germany
{golm, kleinoeder}@informatik.uni-erlangen.de

Abstract

Reflection has attracted considerable attention in recent years. However there are very little systems that fully support reflective programming and are in real use. In this paper we try to understand the reasons for this lacking acceptance of fully reflective systems and explain how our own reflective Java system, metaXa, tries to tackle some of these problems.

1 Introduction

Reflection is a mechanism to gather information about ones own execution environment and to modify this environment if necessary. This reflective computation should be transparent to the base-object computation so that base objects can be reused with different reflective systems.
We explain, how reflection is employed in several parts of the Java system. It becomes obvious that most of these uses of reflection are far more innovative than the well-known Reflection API and would benefit from a unifying reflection mechanism. To illustrate one possible implementation of such a mechanism we describe metaXa[1], our own reflective Java [4] system and try to capture desired properties of future reflective systems. This paper does not discuss class-based reflection or compile-time reflection. We will concentrate on object-based behavioral reflection at runtime [1].
The paper is structured as follows. The first part describes the use of reflection in standard Java. It follows an introduction to the metaXa system. To ameliorate the understanding of reflection we describe the relation between object-oriented programming and reflective programming. Then we discuss some problems that are inherent to reflective programming and still have no satisfying solution.

1. formerly known as MetaJava

2 Reflection in standard Java

Since the introduction of Java more and more reflective features found their way into the system.
A reflective component of the first days is the security manager. It controls access to several vital classes, such as the File and Socket classes. Each time a method of these classes is executed, the security manager is called and asked if the method invoker has the right to execute this method. The security manager can base its decision only upon the information whether the class of the caller object was loaded by the system class loader or by a user-defined class loader.
The second important reflective component for Java as a language for mobile code is the class loader. A class loader transforms an array of bytes (something that the user can analyze and manipulate) into a class. This is a good example of reflection, however the reverse way - reification - is missing. It is not possible to transform a class into an entity that can be analyzed and manipulated by a program, although the reflection API provides a limited way of doing this.
The Reflection API enables structural reflection and was introduced with JDK 1.1. It has only very limited functionality and is mainly used to implement the Beans system and object browsers.
The newest reflective addition to Java, coming with JDK 1.2, are weak references and information about memory utilization. Using this information a program can be cautious when creating new objects in low memory situations.
Object serialization is a meta system that implements persistence. It provides several ways for the programmer to specify information about properties of the base-level objects. The first information is whether serialization of an instance of a class is allowed at all. This is done by (mis)using the interface mechanism and employing the so-called marker interface Serializable whose sole purpose is to annotate the object. The transient keyword is used in a class

definition to annotate those variables that should not be serialized. This is a declarative way — Java provides also a procedural way in form of the Externalizable interface.

The RMI system also employs interfaces for annotation purposes. If a parameter type implements the marker interface Remote it is considered a remote object that is passed by remote reference. All other types are passed by copying. The use of marker interfaces highlights the drawbacks of a missing general purpose annotation system.

An obvious problem with the standard Java approach to reflection is the mangling of reflective and non-reflective code.

3 The metaXa system

When we designed the metaXa system we wanted a reflective system that, besides allowing structural reflection, also provides some way of behavioral reflection, at least for method invocations.

In a reflective system it must be clear and easy to understand:

(1) How the meta system is connected to the base system.

(2) How this connection can be customized.

(3) How the level shift is done.

In most reflective systems there are three types of code:

- the base-level code, that solves the application problem
- the meta-level code, that adapts the base-level code to different environments
- the configuration code, that connects base and meta

Our aim is to separate these three categories of code.

Furthermore we want to retain type safety and efficiency. Our approach to reflection is not language-based, but system-based. Therefore we did not extend the language but built our own extended Java Virtual Machine (JVM).

3.1 Events

Events are the metaXa approach of transferring control from the base level to the meta level. There are events for several JVM mechanisms, such as method invocations, variable accesses, monitor operations, object creation, or class loading. The event handler of the meta object gets an event object that contains the necessary information to perform the requested operation. Event passing is synchronous – this means, that base computation is suspended until the event handler returns.

3.2 Attachments

MetaXa allows stacking of metaobjects (Figure 1). There

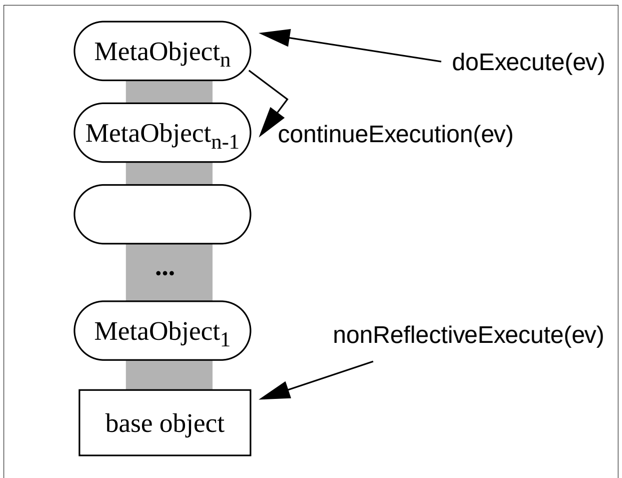

Figure 1

are three ways to activate a mechanism. Instead of invoking a method of another base object the method doExecute can be called. To avoid reflective overlap, the continueExecution method must be used in the event-handling code. The event object (ev) contains the information at which metaobject the method execution continues. In some rare cases, the base-level object must be accessed directly. This is done with nonReflectiveExecute.

These three ways of activating a mechanism are also used for other reified mechanisms. For example there are doReadVariable, continueReadVariable, and nonReflectiveReadVaribale methods.

Metaobjects can not only be attached to objects but also to other interesting parts of the virtual machine (see Figure 2). Metaobjects can also be stacked at these base entities.

- *Class loader*: Attaching a metaobject to the system class loader enables us to use different strategies for class loading, depending on the class name and the object that needs the class.

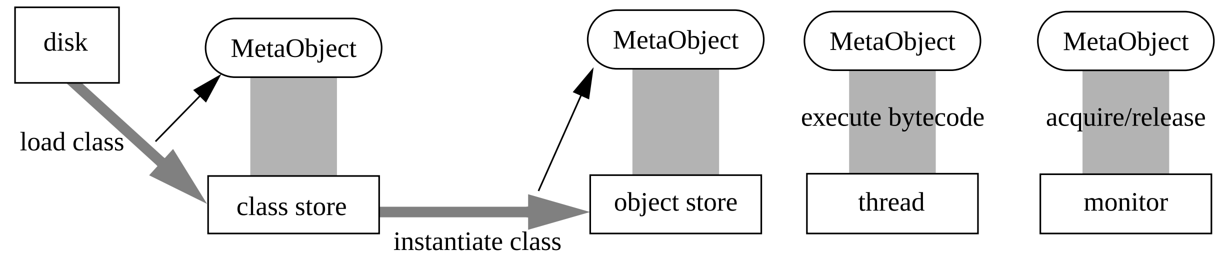

Figure 2

- *Monitors*: Attaching metaobjects to monitors allows us to use different locking policies (e.g., priority based lock allocation or no locking at all).
- *Object creation*: The control of object creation allows attaching metaobjects to all objects of a specific class. It allows creating objects differently from the standard way.
- *Threads*: A metaobject can be attached to a thread to receive an event if a specific opcode, for example invokevirtual, is executed. The metaobject can implement the opcode differently from the VM implementation as long as it changes the stack in the same way the VM implementation does.

3.3 Metaobject propagation rules

Wherever a new object is created, a metaobject may have to be attached. If an attach statement was written at all these places, attachments would be scattered throughout the whole program and become unmanageable. We solve this problem with *propagation rules*. We separated base level and meta level using events and we use the same principle here to separate base-level code and configuration code. Propagation rules describe, which events cause additional attachments of which metaobjects to which base objects. This way events can implicitly cause configuration changes. Let us look at an example of a persistence metaobject. An object should be written to disk after a method execution if this method execution changed the state (the value of its instance variables) of the object. All objects that are referred by this object should also be written to disk. During the object's lifetime these references change. If the object acquires a new reference, a metaobject is attached to the new reference. If it gives up a reference, the persistence metaobject is detached. The propagation rule reads as follows: *If a reference rx to object x is written into an instance variable and overwrites reference ry to object y, then detach metaobject m from y and attach it to x.*

3.4 Annotations

Annotations provide the meta system with information about the base system [5]. To get an understanding of how a general purpose annotation mechanism must work we investigated what kind of information about the base objects is needed by metaobjects (MO) that reify method invocations.

- All metaobjects need the name of the method and a reference to the receiver object (this).
- Security MO: A metaobject that replaces the security manager needs to know information about the caller. If it shall only mimic the standard SecurityManager it needs to know if the caller class was loaded by a classloader or from the local trusted classes.
- Remote Invocation MO: A metaobject that forwards method invocations needs to know information about the marshalling strategy - should parameters be passed by value or by remote reference? If they are passed by value - how deep should the object graph be copied? Has a consistency protocol to be installed (replication) or should the objects just be copied?
- Persistence MO: When should the object be written to disk? Before or after a method execution? Before or after instance variables have been accessed? Only together with other objects?
- Replicated Invocation MO: The base system must provide information about how much consistency can be relaxed. When replication is used for fault tolerance, the fault-tolerance meta system must provide information about the minimal number of replicas and the replica placement. If replication is used for load balancing, the load-balancing meta system must collect load information and decide on the replica placement.

In the current metaXa system all this information must be passed to the metaobject during instantiation of the metaobject or later through configuration calls. There is no language support for annotations, nonetheless a future system would benefit from such a support.

3.5 The meta-level interface

A metaobject accesses the virtual machine through a meta-level interface (MLI). The MLI contains methods for structural reflection. Furthermore there are methods to activate the JVM mechanisms as described in Section 3.2. The MLI also contains some auxiliary methods, for example to create an object that's state (its instance variables) is completely managed by the metaobject. No memory is reserved for such an object by the JVM. This creation mechanism is useful for sparsely populated arrays or for objects that serve as stubs or proxies.

3.6 Advantages and shortcomings of metaXa

The metaXa system has several advantages. It uses a popular language and therefore can soon rely on large libraries and sophisticated development environments. Due to the separation of base-level code and meta-level code one can start with normal programming and gradually introduce metaobjects.
The advantages of metaXa can be summarized as follows:

- There is an n-to-n relation between classes and metaobjects of class instances. Several reflective systems connect the class of the base-level object and the class of the meta-level object in a very early development stage. In such systems it is not possible to use different metaobjects for base-level objects of the same class. The metaXa system allows modification of the metaobject relation at run time.
- The metaobject has complete control over argument and return value passing. As long as it adheres to the static typing rules it can pass every data it wants as parameter or return value.

- It can be controlled, whether a certain operation is performed, how it is done, how it is delayed (e.g. by blocking the thread) or rejected (e.g. by throwing an exception).

On the other hand, metaXa has some disadvantages. The greatest disadvantage is the missing language support for reflective programming. During configuration of the metaobjects and communication through the MLI, the programmer must refer to certain entities of the programming model. This could be first class entities, such as objects, but also other entities (like methods or slots), which can only be referred to indirectly using strings. If the compiler could check the correctness of these references, several programming errors could be detected in an early development stage. A very common error when programming the metaXa system are invalid method references that are caused by typos in the method-name string.

4 Reflection and OO programming

Reflection introduces a new kind of polymorphism in object-oriented programming. The power that comes with polymorphism must be carefully controlled. As subtype polymorphism forces the programmer to define semantically compatible subtype implementations, reflective polymorphism requires the programmer to use only metaobjects that are semantically compatible to the already attached metaobject or to the VM implementation of the mechanism. Semantic conformance is at least partially a matter of personal judgement and can not generally be verified by tools.

As the inheritance relation allows to transparently substitute different implementations that conform to a common semantics, the metaobject relationship allows to substitute different execution environments, that conform to a common semantics. One could consider this conformance as the *principle of least surprise*. It means that a method-call mechanism is replaced always by another method-call mechanism (e.g. a remote call, replicated call, traced call, or delayed call) and not by a mechanism that, for example, deletes the caller or callee object.

	OO	**Reflection**
Extensibility	Subclassing	Metaobject change
Behavioral change	specialized method implementation (overriding)	specialized runtime environment for base-level object
Abstraction	using base classes	thinking in terms of the default execution environment
Referring to a replaced behavior	invoking “super”	passing the event down the metaobject chain

5 Problems in reflective programming

The programming problems that come with reflection must be understood. The following incomplete list highlights some of these problems:

- *Attachment killer*: If one compares a reflective reference with a non-reflective one and then uses the non-reflective reference, attachments can effectively be deactivated. If such a situation occurs, there probably exists a configuration error. This error could be corrected by either attaching the metaobject to the object instead to the reference or better by controlling the flow of references.
- *Reflective overlap*: A common source of reflective overlap are debugging messages in the metaobject’s method-event handler, like `System.out.println(“Base object: “+base)` which invokes the `toString` method at object `base`. Another situation of reflective overlap happens when attaching to a thread and executing the metaobject in the same thread. In metaXa a different thread is used in this case.
- *Compiler optimizations*: Some optimizing compilers remove private methods by inlining them. So it is not possible to reify these methods.

One topic of future research must be the development of programming techniques for reflective systems. Design patterns for reflection could help to solve the composition problems of meta systems. On the other side, reflection makes many behavioral patterns superfluous. Especially most of the patterns that are used to reify message passing, e.g. the Proxy pattern.

5.1 Trade-offs

There are some contradicting requirements when designing a meta architecture:

- Errors should be detected very early by testing typical use cases of the program, *but* reflective programs should adapt to unanticipated changes of the execution environment.
- The VM should be open for extensions by metaobjects, *but* the VM should be robust. For example, if it is possible to modify bytecodes and install them again, robustness should be assured through bytecode verification. There are cases where guaranteeing robustness is no longer possible. We currently develop a native code compiler that can install arbitrary native code. This means that this metasystem can crash the whole VM and therefore it must be a trusted and carefully tested component.

5.2 Limiting reflective power

A reflective system must consider several security issues. We adopted the Java security assumption that local classes can be trusted and classes loaded by a classloader are not trustworthy.

Our security approach relies on two points:

- Classes in the meta package can only be loaded from the local disk Thus, remote classes can only access the meta-level interface by subclassing classes of the meta package.
- Class loading can be controlled by a class-loader metaobject. This metaobject can check inheritance relationships of the loaded class. The remote class can be restricted to subclass only meta-package classes with limited power.

In the initial version of metaXa every metaobject had access to the MLI. To make the system secure, we separated metaobject and MLI. A metaobject can receive events and can be attached to base-level objects. The MLI is accessed through an *mli* reference. Attachment itself is also done via the mli reference. The meta-level programmer can decide who gets the mli reference how much can be done with this particular reference. So this reference serves as a capability.
A problem since the first version of Java was how to store information produced by an applet without compromising client security. Our solution is based on a MetaPersistence metaobject that can be attached to objects to make them persistent. This metaobject has only limited reflective power: It can only read and write slots of objects it is attached to. And it can only be attached to instances of classes loaded by the same classloader as itself. Furthermore it can only write to a limited area of the disk and has a limited amount of space available on this disk. The superclass MetaPersistence counts bytes when writing to the disk.
Furthermore a metaobject can only register events at base-level entities it is attached to. This means that controlling attachments is sufficient to control the JVM mechanisms that metaobjects can modify.

5.3 Expressiveness

The expressiveness or completeness of a reflective system can be estimated in the number of questions about program behavior that could be answered using the reflective system. Examples of such questions are

- On which way did the control flow reach one specific method?
- On which way was a reference passed to a specific object?
- Who is currently the owner of a specific lock?

Sometimes the information is available (e.g. control flow can be deduced from the stack) but in other situations extra computations must be performed. These extra computations should not affect the performance of the rest of the system.

5.4 Testing and Debugging

Although a meta system can be a useful aid in debugging base-level code, the metaobjects themselves must be tested and debugged. Testing a system that uses metaobjects is not easy. Testing means running through almost all path's through a program. This is very difficult in object-oriented programs that use subtype polymorphism and nearly impossible for programs that use reflective polymorphism.
When it comes to testing and debugging a programmer must be able to understand what's going on during a program execution. If he knows how metaobjects are attached to base objects and what the meta system is expected to do, it should be easy to figure out how the complete system behaves.

5.5 Profiling

Profiling is a method to identify performance problems. As every other part of a system also metaobjects can be the reason for performance problems. One way to profile a single-threaded application is to attach a metaobject to the thread that executes the application. This metaobject then counts the number of metalevel executions and measures the time spent for base-level computation and for meta-level computation.

5.6 Performance

With some notable exceptions [7][6][8], performance problems are rarely addressed and never completely solved in reflection research. But to become accepted by a wider community we must prove that reflection can be implemented in a way that does not affect performance too much. In fact, in our current work with just-in-time compilers we try to prove that reflection can actually be used to *improve* overall system performance.
We described the solution to several performance problems elsewhere [2]. Where applicable we use dynamic code generation to modify the behavior of base objects.

6 References

[1] J. Ferber. Computational Reflection in class based Object-Oriented Languages. *OOPSLA ‘89*, pp. 317–326.

[2] M. Golm. *Design and Implementation of a Meta Architecture for Java*. Diplomarbeit (masters thesis). Jan. 1997.

[3] J. Kleinöder, M. Golm. MetaJava: An Efficient Run-Time Meta Architecture for Java. *IWOOOS ‘96*, 1996.

[4] T. Lindholm, F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Sept. 1996.

[5] K.-P. Löhr. Concurrency Annotations for Reusable Software. *CACM* Vol 36 No. 9, Sept. 1993.

[6] C. Zimmermann V. Cahill. It's Your Choice - On the Design and Implementation of a Flexible Metalevel Architecture

[7] J. McAffer. Engineering the Meta Level. *Proceedings of Reflection ‘96*, San Francisco, CA, April 1996.

[8] Y. Yokote. Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach. *Workshop on Reflection and Meta-level Architectures at OOPSLA 93.*