Mirrors: Design principles for meta-level facilities of object-oriented programming languages

CONFERENCE PAPER · JANUARY 2004

DOI: 10.1145/1028976.1029004

2 AUTHORS:

Gilad Bracha
Google Inc.
51 PUBLICATIONS **4,093** CITATIONS
SEE PROFILE

David Ungar
IBM
100 PUBLICATIONS **4,657** CITATIONS
SEE PROFILE

Mirrors: Design Principles for Meta-level Facilities of Object-Oriented Programming Languages

Gilad Bracha
Sun Microsystems
4140 Network Circle
Santa Clara, CA 95054
(408) 276-7025
gilad.bracha@sun.com

David Ungar
Sun Microsystems
2600 Casey Ave., MTV 29-XXX
Mountain View, CA 94043
(650) 336-2618
david.ungar@sun.com

ABSTRACT

We identify three design principles for reflection and metaprogramming facilities in object oriented programming languages. *Encapsulation:* meta-level facilities must encapsulate their implementation. *Stratification:* meta-level facilities must be separated from base-level functionality. *Ontological correspondence:* the ontology of meta-level facilities should correspond to the ontology of the language they manipulate. Traditional/mainstream reflective architectures do not follow these precepts. In contrast, reflective APIs built around the concept of *mirrors* are characterized by adherence to these three principles. Consequently, mirror-based architectures have significant advantages with respect to distribution, deployment and general purpose metaprogramming.

Categories and Subject Descriptors

D.3.2 [Language Classifications]: Object-oriented languages.

General Terms

Design, Languages.

Keywords

Reflection, Metaprogramming, Mirrors, Java, Self, Smalltalk.

1. INTRODUCTION

Object-oriented languages traditionally support meta-level operations such as reflection by reifying program elements such as classes into objects that support reflective operations such as getSuperclass or getMethods.

In a typical object oriented language with reflection, (e.g., Java, C#, Smalltalk, CLOS) one might query an instance for its class, as indicated in the pseudo-code below:

```
class Car {...}
Car myCar = new Car();
int numberOfDoors = myCar.numberOfDoors();
Class theCarsClass = myCar.getClass();
Car anotherCar = theCarsClass.newInstance();

Class theCarsSuperclass = theCarsClass.getSuperclass();
```

Looking at the APIs of such a system, we expect to see something like:

```
class Object {
 Class getClass();
...
}
class Class {
 Class getSuperclass();
// many other methods: getMethods(), getFields() etc.
}
```

The APIs above support reflection at the core of the system. Every object has at least one reflective method, which ties it to Class and (most likely) an entire reflective system. Base- and meta-level operations coexist side-by-side. The same class object that contains constructors and static attributes also responds to queries about its name, superclass, and members. The same object that exhibits behavior about the problem domain also exhibits behavior about being a member of a class (getClass).

This paper argues that meta-level functionality should be implemented separately from base-level functionality, using objects known as *mirrors*. Such an API might look something like this:

```
class Object {
// no reflective methods
...
}
class Class {
// no reflective methods
...
}
interface Mirror {
String name();
...
}
class Reflection {
public static ObjectMirror reflect(Object o) {...}
}
interface ObjectMirror extends Mirror {
 ClassMirror getClass();
...
}
interface ClassMirror extends Mirror {
 ClassMirror getSuperclass();
...
}
```

In a mirror-based system, one might write our example as:

```
ObjectMirror theCarsMirror = Reflection.reflect(myCar);
ClassMirror theCarsClassMirror = theCarsMirror.getClass();
ClassMirror theCarsSuperclassMirror = theCarsClassMirror.getSuperclass();
```

At first glance, the change does not seem to have made much difference. However, the changed API has the effect of:

- divorcing the interface to meta-level operations from a particular implementation, and
- pulling meta-level operations out into a separable subsystem.

Each of these properties manifests an important design principle. The former embodies the principle of *encapsulation*: ***meta-level facilities must encapsulate their implementation***.

The latter corresponds to the principle of *stratification*: ***meta-level facilities must be separated from base-level functionality***.

Another principle is *structural correspondence*: ***the structure of meta-level facilities should correspond to the structure of the language they manipulate***. Any meta-level language architecture that respects these principles is, by definition, a *mirror-based system*. In addition, we advocate the principle of *temporal correspondence:* ***meta-level APIs should be layered so as to distinguish between static and dynamic properties of the language they manipulate.*** These two principles can be coalesced into a broader principle of *ontological correspondence*: ***the ontology of meta-level facilities should correspond to the ontology of the language they manipulate.***

We will show that adherence to the aforementioned design principles yields significant advantages with respect to distributed development and application deployment. We further argue that a well designed mirror-based reflective API can serve as a general purpose metaprogramming API.

Figure 1 illustrates a traditional reflective design and a mirror based one. In a traditional API, classes straddle the boundary between the base level and the meta level. In a mirror based design, one moves from the base level to the meta levels by means of a reflect operation. The levels are clearly separated. In fact, the presence of classes at the base level is not strictly necessary.

The principle of encapsulation is a basic rule of software engineering, yet, as we will show, in many cases it has not been applied to the design of the reflective architectures built-in to major programming languages. The principle of stratification is well known in the reflection community [Maes87], but again has not been consistently adhered to in most programming languages. Structural correspondence was elucidated by, e.g., [34]; while it is substantially respected, we highlight violations and their implications for mainstream languages. Temporal correspondence is related to the well known distinction between compile-time, load-time and run-time reflection. These phases are not always applicable to a given language, but even when they are, only run-time reflection is usually supported by the language.

This paper is the first systematic discussion of the design principles of mirror-based systems and the concomitant advantages. The advantages of mirrors include:

- The ability to write metaprogramming applications that are independent of a specific metaprogramming implementation such as reflection.

  With care, metaprogramming clients can interact with metadata sources that are local or remote without any change to the client. Furthermore, a client can interact with multiple sources of metadata at run time, and in fact interact with metaobjects from different implementations simultaneously.
- The ability to obtain metadata from systems executing on platforms that do not themselves include a full reflective implementation. Examples:
  - Small, memory-constrained devices or embedded systems
  - Deployed applications where concerns of footprint, security or bandwidth have discouraged or precluded the deployment of built-in reflection support.
- The ability to dynamically add/remove reflection support to/from a running computation.
- The ability to deploy non-reflective applications written in reflective languages on platforms without a reflective implementation, reducing footprint or saving communication time.

Terminology. Reflective language architectures may be characterized in terms of their support for:

1. **Introspection**. The ability of a program to examine its own structure.
2. **Self-modification**. The ability of a program to change its own structure.
3. **Execution of dynamically generated code**. The ability to execute program fragments that are not statically known. This is a special case of 2.
4. **Intercession**. The ability to modify the semantics of the underlying programming language from within the language itself (the term intercession is sometimes used in a broader sense in parts of the literature, but we adopt the narrower definition of intercession given here, based on [18]).

Table 1 summarizes the support for these features in several reflective systems mentioned in this paper.

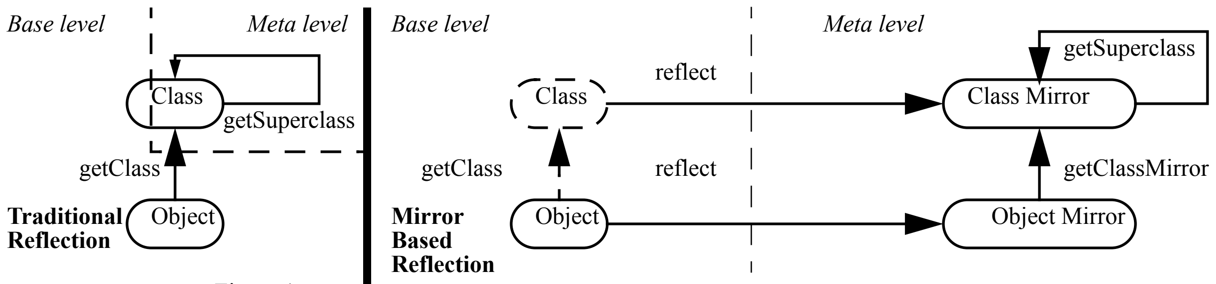


Figure 1

Table 1

	Introspection	Self Modification	Intercession
Java Core Reflection	Yes	No	No
Smalltalk	Yes	Yes	Very limited
CLOS	Yes	Yes	Yes
JDI	Yes	Limited	No
Strongtalk	Yes	Yes	No
Self	Yes	Yes	Very limited

The term *reflection* refers to situations where a program manipulates itself. We use the more general term *metaprogramming* to describe situations where a program manipulates a (possibly different) program. The word *program* itself is often used to describe two distinct notions: a description given in some programming language and an executing computational process. We shall refer to the former as *code*, and to the latter as *computation*.

Each of the next three sections focuses on one of the three principles identified above. In each section, we show concrete problems that stem from violations of the principle being discussed, and how they can be solved using mirrors. The following section discusses the principle of encapsulation and its implications for distributed execution. This leads to the need for stratification, discussed in section 3 alongside issues of deployment. Section 4 then deals with the principle of correspondence and the problems that arise when it is neglected. Section 5 gives an overall discussion of the issues that arise in the design of mirror-based systems. We then discuss related work and present our conclusions.

2. ENCAPSULATION

It is a basic principle of software engineering that a module should not rely on the particulars of another module's implementation. Unfortunately, clients of classical reflective APIs are dependent on implementation details of the reflective system they use. We demonstrate this point with a case study, followed by a more general analysis.

2.1 Case Study: Distribution

Consider the following scenario. A programmer writes a class browser, using reflection. At a later time, it becomes necessary to browse classes on remote machines. We would like to reuse as much of our browser code as possible, with as little adaptation as possible.

This goal may seem controversial; readers should note that we are not arguing for (or against) transparent distribution. That controversy is far outside the scope of this paper. Rather, we argue that mirrors are a good approach to the design of a reflective API that *is* distribution-aware. We also claim that a mirror-based, distribution aware reflective API can be designed so that it serves for the non-distributed case as well.

With the scenario given above in mind, we contrast two APIs that are based upon the same programming language and virtual machine - Java core reflection and the Java Debugger Interface (JDI).

We first present an overview of Java core reflection. Java core reflection is the reflective API provided as part of the J2SE and J2EE platforms. It is a traditional reflective API rather than a mirror-based one. Next, we'll see how core reflection deals with the scenario described above. This is followed by a brief introduction to JDI, a mirror-based API designed to support debugging of Java programs. We then return to our example and show how JDI facilitates a satisfactory solution.

2.1.1 Java Core Reflection

In the Java programming language [16], reflective capabilities are centered around class java.lang.Class (Class for short), extended by the core reflection package java.lang.reflect. All classes, including Class itself, are instances of Class.

In addition to Class, several other classes that support reflection are defined, e.g., java.lang.reflect.Method, java.lang.reflect.Field, java.lang.reflect.Constructor. The reflective API is defined using classes, rather than interfaces. The significance of this is discussed in section 2.2 below.

Classes have methods that support introspection. One can query a class for its superclass, superinterfaces, fields, methods, constructors and member classes. It is not possible to alter the structure or code of an existing class using these facilities, as they do not support self-modification.

2.1.2 Applying Core Reflection to the Class Browser Problem

It's quite easy to write a browser that, given the name of a class, will allow us to examine the class' structure using core reflection. However, once we try and use the same code on remote classes, we encounter serious difficulties [26].

Since Java core reflection does not directly support distribution, the browser will need to use an alternate implementation. How do we avoid the need for a complete rewrite of the browser?

One possibility, following [20], is to design the distributed metaprogramming API to have the same class and method structure as core reflection. Then the browser application can switch between the local and distributed APIs by changing a single import statement, from import java.lang.reflect.* to, say, import com.mycompany.distributed.metaprogramming.*.

This approach is problematic. We must maintain two (slightly different) copies of the source code. We have two sets of binaries to distribute and load. Consequently, at runtime, our browser code has twice the memory footprint, and it is very hard for the two loaded versions to interoperate. Furthermore, if our browser interacts with the class loader API [19], it will need to use class Class and hiding it using import will not be possible.

We may also wish to use the browser to observe source code outside of a running virtual machine. The problems mentioned above are exacerbated; we have three versions of the source, three sets of binaries etc.

Import statements fail to address the problem because they cannot be used to decouple modules; rather, they couple them in a localized fashion.

Another set of difficulties is specific to distributed programming. Our application should be able to deal with network failures and latency, and this will affect the logic of the application. We will also need to specify and display the network locations of the classes we are browsing.

Altogether, the core reflection API is unsuited to our purpose.

2.1.3 JDI

The Java Debug Interface (JDI) is the uppermost layer of the Java Platform Debugger Architecture (JPDA) [29]. It is designed to support remote debugging, but supports all the introspection capabilities present in Java core reflection as well. JDI also supports limited forms of self-modification.

JDI defines interfaces that describe all program entities that might be of interest to a debugger. These include classes, interfaces, objects, stack frames and more. All these interfaces are subtypes of the interface com.sun.jdi.Mirror. Below we focus on those mirror interfaces that are most important, and on those that involve unique issues not seen in other languages or systems.

A mirror is always associated with a particular virtual machine, in which the entity being mirrored exists. The interface com.sun.jdi.VirtualMachine describes a mirror on a virtual machine (VM) as a whole.

One can obtain the set of loaded classes and interfaces in the VM being mirrored, the set of threads on the VM, information regarding the capabilities of that VM, mirrors for specific classes or values on the VM etc.

Objects are mirrored by objects implementing the interface com.sun.jdi.ObjectReference (this is the equivalent of the ObjectMirror interface shown in the introduction). It is possible to read and write the mirrored object's fields, invoke its methods, and to get its class.

Remote mirrors on objects raise issues of distributed garbage collection. By default, a mirrored object may be collected by the mirrored VM. In other words, object mirrors maintain a weak reference to their mirrored object. It is possible to override this default, and also to determine that an object has been collected.

Threads are mirrored by objects implementing the interface com.sun.jdi.ThreadReference, which is itself a specialization of ObjectReference, since threads are objects in the JVM. Operations on threads include suspension and resumption, and operations on the thread's call stack.

2.1.4 Scenario Revisited Using JDI

Using JDI, it is straightforward to write a class browser that can be used to examine classes in separate processes on remote machines or on the same virtual machine.

JDI was designed with distributed use in mind. Debugging the same virtual machine using JDI is not recommended, because debugging entails stopping threads on the virtual machine being debugged. If JDI is running on that same VM, there is a substantial risk of deadlock.

However, structural introspection does not usually require threads to be paused. Furthermore, even if the usual JDI implementation were not well-behaved, an alternate implementation can be derived by wrapping core reflection within objects that implement the JDI interfaces as needed to support structural introspection.

Concretely, one can construct different implementations of the interface VirtualMachine (mirroring a particular VM) that support reflection on either the current (local) process or remote processes. A class browser would query an instance of VirtualMachine for loaded classes (represented as a list of ClassType, the mirror interface for classes). The browser code itself is oblivious to the difference between the different implementations of the mirror interfaces.

While we have established that JDI can be used for non-distributed reflection, we have not shown that it is as convenient to use as Java core reflection. The main difficulty is the need to deal with potential exceptions that can arise only in distributed use. While this difficulty should not be ignored, we would argue that it is outweighed by the benefits of having to learn a single API rather than two. Furthermore, as discussed in section 3.1.3, experience with the Self mirror APIs indicates that it is possible to employ the same API for local and distributed reflection without excessive penalty.

2.2 Analysis

Our case study shows that Java core reflection does not support distributed development tools, while JDI does. Partly, this is because JDI deals with certain distribution-specific issues, such as network failure and distributed memory management, while core reflection does not.

These issues could be addressed by an alternate implementation of core reflection that communicated with remote JVMs via proxies. In the event of network failure (or unacceptable latency) operations might fail by throwing an exception.

The key point is that such an alternate implementation is not possible because the core reflection API is based on classes rather than interfaces. Core reflection deliberately violates the principle of encapsulation, by making its clients dependent upon specific implementation types (classes). This dependency is enforced by the type system and prevents clients from using alternate implementations of the core reflection API.

Of course, in a dynamically language one can write a proxy emulating the reflective API without concern for quirks of the type system. However, even in dynamically typed object oriented languages, the implementation of an object may be subtly exposed, as discussed below.

2.2.1 Encapsulating Class Identity

In most object-oriented languages, the getClass() method (or its equivalent) exposes information about the implementation to clients. Applications may come to depend on this information, making it very difficult to replace an object with another one that has equivalent functionality but is an instance of a different class.

A simple example is:

```
if (aCar.getClass() == Car.class) {...}
```

this is very bad practice, yet not uncommon. This code will fail if aCar is an instance of any alternate implementation of Car.

Now consider an example in a language where classes have application specific state and code (as in Smalltalk or CLOS). Class Car might have a method

```
numberOfCarsMadeIn(y) {...} // return the number of cars manufactured in year y
```

typically used as follows:

```
n = aCar.getClass().numberOfCarsMadeIn(1999);
```

If aCar is a proxy for a remote car object, the standard implementation of getClass would return Proxy, causing the code to fail. It is clear that we want getClass to return a proxy on the remote car class. Doing so, however, poses a problem: how is reflective code going to get hold of the real class of aCar, class Proxy? We might define another method, getRealClass, but this merely perpetuates the original problem of exposing class identity. The functionality provided by getClass defeats the very reuse that has been propounded as a motivation for object-oriented programming.[1]

The solution is to factor the reflective functionality of getClass out of the API of ordinary objects. This is exactly what mirrors do.

This factoring implies a functional decomposition rather than a classic object-oriented one. The mirror implementation decides how to mirror objects of a given kind, instead of leaving that decision to the implementation of the objects themselves.

In some scenarios, such as the class browser example, this is quite natural. The browser knows where it is looking for classes - locally, remotely, in a database etc., and can choose a suitable mirror factory. In other cases, such as a debugger on a local process that encounters a proxy object, it isn't immediately clear which mirror to choose. It may be a configuration preference set by the user.

It follows that mirror factories may have to dispatch on the type of the object. How do they do so if access to the identity of the class is denied, as we recommend? The answer is that basic, local reflection inherently does not respect the encapsulation of other objects, and can be used by reflective applications, including other mirror factories, to identify classes if they so choose. One might even define a "public mirror" factory that would allow classes to be registered indicating what mirror implementation to use when reflecting upon their instances.

There are usually several means other than getClass by which the identity of an instance's class may be detected. A common example is the use of constructs like instanceof or checked casts in conjunction with class types (using these constructs with interface types is harmless). Such usage, and any reliance on class identity, should be avoided in application code.

Our conclusion is that separation of mirrors, at the meta-level, and classes, at the base-level, is necessary to fully support encapsulation. This separation is a manifestation of the principle of stratification, discussed further in the next section.

3. STRATIFICATION

A desirable engineering property of a feature is that it not impose any costs when it is not used. Adherence to the principle of stratification supports this desideratum by making it easy to eliminate

[1] This problem may not be so important to those for whom object-oriented programming's modelling abilities are paramount, so perhaps we have to admit that this paper comes at its subject from a non-Scandinavian perspective. However it may be that the stratification we propose is not dissonant with the separation of concepts from phenomena.

reflection when it is not needed. This has important benefits in the context of deployment, as discussed below.

3.1 Case Study: Deployment

When deploying an application, it is not always desirable to deploy it together with all the reflective facilities available in the language. The application may not require these reflective capabilities at all, or it may require them infrequently. In such cases, it may be advantageous to reduce the application footprint by avoiding or delaying the deployment of reflective facilities. This is especially true on small platforms such as mobile phones, PDAs, smart cards or other embedded systems.

Our goal, then, is to avoid deploying reflection unless or until it is actually needed by the application. We now review the Smalltalk-80 reflective API, contrast it with Strongtalk, a mirror-based Smalltalk system, and give an analysis of how such different architectures affect the deployment problem.

3.1.1 Smalltalk-80

Smalltalk-80 differs from most languages in that a program is not defined declaratively. Instead, a computation is defined by a set of objects. Classes capture shared structure among objects, but they themselves are objects, not declarations. The only way to create new classes, add code to classes etc. is to invoke methods upon them. Smalltalk classes inherently support self-modification because reflection is the sole mechanism available for constructing and modifying them. The method class is defined for all objects, so that one can obtain the class of any instance. Every object also implements the inspect method, which opens an inspector on the object.

Smalltalk classes are not used exclusively as meta-objects. Classes typically include application specific methods and state. The most common use of class methods is for instance creation. There is no special syntax for instantiating a class, nor is there any notion of a constructor in Smalltalk. Instead, class methods are used to create new instances.

Because Smalltalk classes play both application specific and meta-level roles in a program, it is generally difficult to remove reflection support from a Smalltalk application. We discuss this topic further in section 3.1.3.

3.1.2 Strongtalk

Strongtalk differs from traditional Smalltalk systems in a number of respects. The most relevant differences for the purposes of our discussion are:

- It adopts the use of mirrors instead of the traditional reflective architecture.
- It has an optional static type system [8],[6] that is based exclusively on interfaces, supporting the principle of encapsulation.
- It is a mixin based system [7][9][5].

The Strongtalk mirror system supports introspection and self-modification. The class Mirror and its subclasses support reflection on mixins, classes, types, methods, global variables, objects, the call stack and individual stack frames (activation records). Invoking Mirror>>on: on an object returns an appropriate mirror object.

Mixins serve as the basic unit of self-modification in the Strongtalk mirror API. Mixins are well suited to this task, because they are stateless (unlike Smalltalk classes), and can therefore be copied freely. Modifications can be made to a copy of a mixin without any effect on the ongoing computation. Only when all modifications are complete is the modified version installed in one atomic operation. Several modified mixins can be installed simultaneously. This batching of modifications improves performance, but it has a more important advantage. A series of modifications may be consistent as a whole, but if done piecemeal, may create inconsistent intermediate versions of the code, possibly leading to program failure. This problem is avoided by the batching the modifications. See [5] for more details.

The usual reflective functionality associated with Class is available in ClassMirror. Similarly, specialized mirror classes exist for mixins, protocols (the rough equivalent of interfaces in Java) and global variable declarations.

Whereas in an ordinary Smalltalk system one might ask a class to remove one of its methods, in Strongtalk one would obtain a mirror on the class using Mirror>>on: and then interact with the mirror, as dictated by the principle of stratification.

To inspect an ordinary instance o, one does not use the inspect method. Instead, one invokes the method Inspector>>launchOn: on the object. This is crucial in decoupling the GUI from the rest of the system.

To determine the class of an object for reflective purposes, rather than invoke its class method, one invokes the method Reflection>>classOf: on the object.

This latter example deserves discussion. In Smalltalk, obtaining an object's class is a routine non-reflective operation. Class methods are used to construct new instances and for other application purposes. For such application specific purposes, the class method can and should be used. Unlike traditional Smalltalks, this method can be overridden in Strongtalk. This allows objects to hide implementation details, including their class. For example, a proxy object can hide the fact that it is an instance of a proxy class. See section 2.2.1 for additional analysis.

3.1.3 Analysis

Mirrors make it easier to eliminate reflective infrastructure from an application. To see why, we must consider the issues in both dynamically and statically typed languages.

In dynamically typed languages that do not use mirrors, it can be difficult to separate reflective facilities and the development environment from the application. For example, the ability to add new methods requires access to a source code compiler. If this capability is placed in class Class, it becomes difficult to weed it out of an application, as all applications rely on class Class. Similarly, in Smalltalk Object>>inspect tends to bind object inspectors and a UI framework into the application.

In general, if reflective capabilities are part of a class that has uses other than reflection, it is hard to safely remove those reflective capabilities from the system. To be sure that an application does not use reflection one needs to resort to sophisticated and costly type inference techniques [2].

Mirrors eliminate this problem by clearly separating reflective functionality, and moving it into places that ordinary applications will not access. It is then straightforward to establish that an application does not require functionality from the reflective subsystem or from the development environment. If the application makes no reference to entry point(s) associated with reflection (e.g., classes Mirror and/or Reflection in Strongtalk, the com.sun.jdi.Mirror interface in JDI, or the reflect: method in Self [33][27]), reflection support can be removed

In statically typed languages that employ a nominal type system, eliminating reflective functionality from an application prior to deployment is considerably easier than in dynamically typed languages. However, if one does not use mirrors, but wishes to avoid deploying the reflective subsystem unnecessarily, one must statically determine that reflection will not be used anywhere in the application. If reflection is not deployed initially, it will not be possible to modify the existing representations of classes, methods etc. to support it afterwards (since one would need self-modification capabilities to do so). This is a real liability in the presence of dynamic loading. Using mirrors, one can add or remove the reflective capacities at run time without special support, using dynamic class loading and unloading. The ability to dynamically enable or disable reflection support is useful from a security perspective as well. Of course, reflection cannot be deployed dynamically without some degree of support from the underlying implementation.

The capacity to reflect on a computation that does not contain a reflective API is demonstrated in Klein, a metacyclic Self VM being developed by the second author. Klein itself does not support a reflective API. Klein is debugged using a Self GUI running on the standard Self VM in a separate process. The GUI communicates with the Klein VM using mirrors that communicate over sockets. No changes were made to Self's mirror API - only a new implementation was needed. This experience supports our contention (in section 2.1) that a single mirror API can serve for both the distributed and local cases.

Overall, we conclude that mirrors facilitate deployment. The advantages are more pronounced for dynamically typed languages, but mirrors are advantageous even when a static type system is used.

4. ONTOLOGICAL CORRESPONDENCE

4.1 Temporal Correspondence

Reflection is traditionally defined with reference to a computation. Naturally then, an underlying assumption of reflective APIs is that reflected entities exist within an executing context. These APIs therefore support operations such as instantiating a class, or querying it for all its instances. While some reflective applications (e.g., profilers and debuggers) actually manipulate a computation, others, such as compilers, class hierarchy browsers and pretty printers, only manipulate the structure of a program (code).

It is desirable to run applications in this latter category on code that is not embedded in a computation. A class browser might be used to view a source database, for example. Conversely, some metaprogramming tools may assume the availability of source information that may be unavailable at run-time. For example, Javadoc [17] expects comments to be available.

4.1.1 Case Study: Browsing via a Source Database vs. Browsing via Reflection

If one writes a class browser using Java core reflection, one cannot easily retarget the application to browse classes described in a source database. The situation is similar to what we encountered in section 2.1.2. We cannot create an alternate implementation of the API that produces instances of Class, java.lang.reflect.Method etc. simply by reading source code without compiling and loading the classes into a running JVM. This is yet another example of the importance of the principle of encapsulation, but there are additional issues involved here.

Even if an alternate implementation of core reflection were possible, we would face difficulties. The reflection API allows methods to be invoked, classes to be instantiated etc. These operations make no sense when the browser is examining a source database.

We would fare no better using JDI, which was designed primarily for debugging. It assumes that there has to be a running VM containing threads, from which one may obtain stack frames, objects and classes. We can see that adhering to the encapsulation principle is a necessary but insufficient condition to solve our problem.

Note that the JDI subset concerned with structural reflection on classes is just as applicable to classes whose structure is extracted from source code or from binary class files. If those elements of JDI that do not depend on the presence of a computation were factored out into a separate API, an implementation that operated upon a source database would be straightforward.

This leads to the following observation: ***mirroring code and mirroring computation should be separable modules of the mirror API***. This is a manifestation of the principle of temporal correspondence. The distinction a language makes between code (compile-time) and computation (run-time) should be manifest in its metaprogramming APIs.

The notion of code is useful for restarting programs in a fresh state, for proving program properties, and especially for transporting programs between processes, as discussed below.

4.1.2 Distinguishing Code and Computation in Self: Interchange of Programs and Data

The Self system strives to harness people's intuitions about the real world to help them program computers. Since the real world does not distinguish code and computation—there is no compile/run switch in the world—Self attempts to unify program and computation. A Self program is just a set of objects, and its mirrors reflect that world view. So, one might argue that the principle of temporal correspondence is irrelevant for Self.

However, Self features the transporter, a system designed to move "programs" (sets of slots containing data or code) from one Self world of objects to another [32]. In building the transporter, the second author was forced to see that there was a need for a program, something that could be described and moved into another world that would provide it with the new functionality. The objects added to the system to represent programs, (annotations, modules, etc.) are indeed meta-level objects that truck in code instead of computation. Despite our own best intentions, when it came time to share programs, we found that this principle applied after all.

4.2 Structural Correspondence

Structural correspondence implies that every language construct is mirrored. This principle has long been recognized in the reflection community [23]. Nevertheless, in practice it is often violated. We discuss some of the issues that arise below.

4.2.1 Reifying Both Code and Computation

Meta-object protocols ideally introduce a meta-object for every object in the computation. However, in many languages, important notions like modules, import and export statements, metadata, types and comments exist only at compile time. Such constructs are liable to be excluded by a MOP that only reifies elements of the actual computation. Similarly, compile-time MOPs deal only with compile-time constructs; the MOP is not present at run-time, and cannot reify entities that exist only at run-time (see section 6.4 for further discussion of compile-time MOPs).

4.2.2 Mirroring Method Bodies

In most languages, constructs below the method level, such as statements and expressions, do not have corresponding meta-objects. This is also the case in the mirror systems we have discussed. At the VM level, byte codes are often available, and these can often be mapped back into source code. This strategy is typically used by tools such as debuggers.

True structural correspondence would imply that a higher level representation of method bodies should be available. This would be useful, so that tools that manipulate source code, such as compilers, could use a standardized representation.

Because source code (or even byte code) may not always be available, many implementors have shied away from providing such facilities. However, it is often possible to provide such functionality conditionally (i.e., if it is available) and/or on-demand. For example, in JDI, clients can query a VirtualMachine about what kinds of operations it supports. This enables JDI to define an API to access a method's byte code, but allows for implementations that do not retain byte code as well.

4.2.3 Which Language to Reflect?

Programming languages that support reflection are often implemented on top of a virtual machine (e.g., Java and the JVM, C# and the CLR, etc.). One must not confuse the metaprogramming API for the language of the underlying virtual machine (the VML) with that of the high level language (HLL) running on top of it. When designing a metaprogramming API, it is important to be clear what the base-level language is. This is true regardless of whether the API in question is mirror-based.

Reflection has to be supported by the VM at some basic level, so a reflective API to the VML is a given. High level languages implemented on top of a virtual machine should ideally include their own reflection API. Maintaining a distinct reflective API for the HLL is valuable for a number of reasons.

The VML reflective API may not maintain the invariants of the HLL, thereby introducing potential security and correctness problems.

There is a risk of discrepancies between the VML and the HLL. Such discrepancies often arise when implementing high level constructs that are not directly supported by the VM.

A prominent example are nested classes in the Java programming language. Implementing nested classes requires the generation of synthetic classes, interfaces, methods and fields that are not present in the original source code. In some cases, constructors may require additional parameters.

Such features should be hidden from HLL programs because they expose the details of a specific translation scheme between the HLL and the VML. Such a translation scheme is an implementation detail that HLL programs must never rely on. In particular, these details should not leak through the reflective API.

As a counter-example, consider java.lang.Class.getMethods, which returns the methods declared by a class. All the methods declared at the VM level are returned, regardless of whether they are synthetic. This exposes the translation strategy of the Java compiler to clients of reflection.

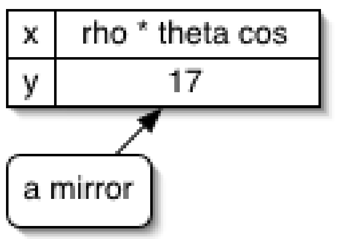
If multiple source languages are implemented on a given virtual machine, the risk of discrepancies among the virtual machine language and the various source languages increases. A common example is method overloading, typically supported by the HLL compiler but not by the underlying VM. If two languages have different overload resolution schemes, a single reflective API will support only one of them correctly.

Even if the HLL and VML are in complete agreement, it is likely that discrepancies will arise over time as new HLL features are added and implemented by the HLL front end without VM support. Again, both nested classes and generics in the Java programming language are examples of this.

To avoid such difficulties, the Strongtalk mirror API is subdivided into high level mirrors and low-level mirrors. High-level mirrors reflect Smalltalk, and low level mirrors reflect the underlying structures in the virtual machine. This distinction is not present in any other reflective API that we are aware of.

High level mirrors are defined by the Mirror class hierarchy. High level mirrors support Smalltalk level semantics. Low level mirrors are defined by the class VMMirror and its subclasses. VM mirrors manifest representational differences between different kinds of objects (e.g., integers, arrays, classes, mixins, regular objects) that are hidden at the language level. One can ask a ClassVMMirror for the physical size of its instances, or the size of their header, for example. The low level mirror API is inherently sensitive to the design of the underlying virtual machine language and implementation.

We conclude that: ***there should be distinct reflective APIs for each language in a system,*** in particular for the underlying virtual machine language and for each high level language running on top



A Self objects with two slots, x and y. Sending y to this object returns 17, sending x to it returns the product of rho and cos theta. There is no way in the base Self language to obtain a reference to this method.

A mirror on the object above. Sending the size message returns 2, sending "at: 'y'" returns a mirror on 17, and sending "at: 'x'" returns a mirror on the method in the x slot.

Figure 2

of the virtual machine. This is an instance of the principle of structural correspondence.

5. ISSUES IN THE DESIGN OF MIRROR-BASED SYSTEMS

5.1 Classes vs. Prototypes

5.1.1 Self

Mirrors were first introduced in the Self programming language[33]. Self uses prototypes instead of classes, but unlike Actors[3], it unifies access to state and behavior. Lacking direct references to methods, the Self language could not support traditional, integrated reflective operations. Self's omission of direct method references stems from its unification of method invocation with variable access and assignment as shown in figure 2. Consequently, there is no way in Self to refer to a method, as opposed to the result of its execution!

The designers of Self felt that method references were not object oriented, because a method does the same thing whenever it is invoked, unlike a message sent to an object where the object gets to decide. However, when it came time to build a programming environment, it became clear that some way would be needed to refer to methods.

The solution was the "mirror." Named originally both as a pun on "reflection" and also to suggest "smoke and mirrors", the original notion of mirror was an object that would appear to be a dictionary whose entries were named by the slot names of the original object (the "reflectee") and whose entries contained mirrors on the contents of the slots of the reflectee, thus satisfying the principle of stratification. Later, "slot objects" were introduced. When asked for the dictionary entry for a given slot, a mirror returns an object that represents the slot. It contains the slot name, attributes such as whether it is a parent slot, and a mirror on the contents of the slot. Self's mirrors support introspection and self-modification.

Here are some examples:

(reflect: anObject) size — returns the number of slots in an object

(reflect: anObject) do: [|:s| s printLine] — prints out the slots in an object, one per line

(reflect: anObject) at: 'newSlotName' Put: (reflect: 17)— adds a slot containing 17 to the object

((reflect: anObject) at: 'fred') setParent: true— turns the slot named "fred" into a parent slot

As Self codesigner Randall B. Smith has observed, the down side of mirrors is a decrease in uniformity: sometimes it is not clear whether a new method should accept a mirror or a base object as its argument. Worse, some functionality, such as printing, does not seem to cleanly fall into either base- or meta- levels. On the whole, though, the designers of Self are quite pleased with how their stratified mirror design has worked out.

In this paper, we argue that mainstream class-based languages benefit from a model of metaprogramming that follows three principles. However, if one accepts the premise in this paper, then one must realize that there is a fundamental problem with class-based languages as we know them.[1] Every single class-based language we know of displays the problems associated with instanceOf and class identity tests as described elsewhere in this paper. We believe that the class-based mindset itself drags along the implication that the class of an object is a reasonable thing for client code to know. But, this very knowledge inhibits reuse. On the other hand, existing prototype-based languages, even Self, do not seem to allow for sufficient latitude for the programmer to express his or her intentions at the linguistic level. Consequently, we agree with Ole Lehrmann Madsen's view [21] that the next important OOPL will bring classes and prototypes together. In other words, we know some of its characteristics but lack a concrete example.

5.2 The Role of Types

We distinguish between

1. Structural type systems.

2. Nominal type systems based exclusively on interfaces.

3. Optional type systems.

4. Dynamically typed systems.

All of these approaches support the principle of encapsulation, but only a mandatory version of (2) can reliably help us identify when the reflective API is not actually being used, thereby supporting our goals for deployment.

A mandatory nominal type system that avoids implementation types can shield designers from some of design errors that affect current mainstream reflective architectures. We believe that such a system is a good choice for languages that seek to employ mandatory typechecking. However, many considerations impact the design of a language's type system, and discussing them is well beyond the scope of this paper.

Fortunately, careful mirror-based design means that one need not rely on the type system to separate out the reflective API. The benefits of a mirror-based API can be had almost independently of the

[1] Of course, there may be other problems with prototype-based languages as we know them.

type system used, if any. The key constraint on the type system is that it avoid relying exclusively on implementation types.

5.3 Designing Languages in tandem with Reflection

By definition, a reflective API reifies the ontology of a programming language. The principle of structural correspondence demands that each construct in the language map to an interface in the API. Examining JDI, we see a large framework that has to reify a complex language ontology (primitive types, classes, interfaces, access control, packages, methods, constructors, initializers etc.). A language's complexity becomes manifest in its reflective API, and the size of the API is directly related to the size of the language. This adds to the attraction of simple languages, with a small number of very general constructs, as opposed to complex languages with a large number of highly specialized constructs.

Given that reflection is a necessity in modern applications, it seems plausible to suggest that languages be designed in tandem with their reflective APIs. If the reflective API seems too large and complex, language designers can take this as an indication that the language itself is too large and complex.

5.4 Metadata

The idea of language support for user defined metadata has garnered much attention recently, with its introduction into C#. Such support has now been added to the Java programming language as well [30]. Metadata in this context consists of user specified data attached to elements of the program source, such as class or method declarations. Design of such metadata facilities raises many of the same issues discussed in this paper - specifically, the ability to examine metadata in distributed settings or when the source is not loaded.

Self's mirrors provide a cozy home for its metadata. Originally, Self had no user-specifiable metadata. Later on in the project, it gained the capability for user-level code to associate an arbitrary object (called an annotation) with any object or slot. The Self virtual machine implemented this facility with extra space in its maps, and exposed the annotations through mirrors. Self-level methods in mirrors implemented all of the annotation functionality, such as get- and set- annotations for objects and slots. By providing a first-class place for meta-level operations, the designer who chooses mirrors prepares for the future expansion of reflective capabilities.

5.5 Disadvantages of Mirrors

Mirror-based architectures reify the distinction between base- and meta-level operations. When this distinction is either awkward or ambiguous, the mirrors can just get in the way. For instance, consider a user-interface object that allows a programmer to inspect the slots of an object. Without mirrors, one might expect a prototype such as: `SlotExaminer newOn: anObject`. But in a system with mirrors, one is faced with an awkward choice: if the message takes an object as argument, the slot examiner cannot be used with a proxy object. But if the message takes a mirror as argument, each invocation of the method must suffer the verbosity of the mirror creation operation. In a non-uniform system, each option has drawbacks.

The issue of what protocol to use for an object inspector may seem moot to a true believer in reflection—after all the inspector is reflecting, so send it a mirror and hang the (verbosity) cost—but sometimes the line between base- and meta- level can blur so far there is no distinction left at all. Consider the operation of printing an object. What most of us consider to be a reasonable printed representation does not respect any separation of base and meta. For example, a list object might print as "A List containing (a Car, a Truck)." The first part of the string uses the name of the class of the object (meta-level), but the last part uses the list iteration code (base-level). A mirror-based architecture adds complexity to printing code by introducing explicit level shifts into the code. Where the distinction between base- and meta-level fails to model the problem to be solved, mirrors become a nuisance instead of a help.

5.6 Future Work: The Ultimate Mirror System

This paper envisages a reflection/metaprogramming API that:

- Supports introspection, self-modification and intercession on both code and computation.
- Includes distinct layers for mirroring the virtual machine language and the high level language(s).
- Is clearly separable from the underlying base language, allowing applications that do not use the reflection/meta-programming API to be deployed independently of it.
- Does not assume a particular implementation; rather it transparently allows for local or remote use and demonstrably allows for multiple implementations.

Such a system should support an IDE remotely manipulating a small footprint VM that does not include a full implementation of reflection, such as one found on a PDA or mobile phone.

None of the reflective systems constructed to date fully meets this goal, as one can see in table 1, which highlights lack of support for self modification and/or intercession, and table 2, which summarizes other key properties of the mirror based systems we have discussed in this paper.

In Strongtalk, the API was designed with all of these goals in mind except for compile-time metaprogramming, mirroring below the method level, and intercession. However, development of Strongtalk ceased before the mirror API was fully mature. As a result, no distributed implementation was ever constructed to validate it.

In contrast, JDI successfully implements distributed metaprogramming in production, but it assumes it is operating on a runtime representation, and makes no separation between the virtual machine and the high level languages. Though the JDI interface is designed to fully support self-modification, actual implementations are more restrictive, and intercession support is completely absent.

Self still lacks complete support for VM level language reflection facilities, and does not fully support either fine-grain reflection below the method level or mirror-based intercession. Other than that, it appears to satisfy our criteria.

The question of how to support intercession in a mirror based set-

Table 2

	Compile time	Run time	VML	HLL	Reflects below Method lvl
Strongtalk	No	Yes	Yes	Yes	No
Self	Yes	Yes	No	Yes	No
JDI	No	Yes	Mixed	Mixed	Yes
APT	Yes	No	No	Yes	No

ting is an intriguing one. Rather than speculate we leave it for future research.

6. RELATED WORK

6.1 Pluggable Reflection

The closest work to this paper is [20] in which Lorenz and Vlissides address deficiencies of mainstream reflective systems. The main focus is on the violation of the encapsulation principle. Rather than consider alternate designs for reflection and languages, they concentrate on a pragmatic methodology and tools that ameliorate the problem for users of existing systems. Using patterns and component techniques, they work on reducing the coupling between reflection and its clients. They also note the problem of temporal correspondence (though the terminology differs), but without offering a solution. They do not directly address the other design principles discussed here.

6.2 Declarative Metaprogramming

Declarative metaprogramming [35] makes use of declarative languages for metaprogramming. In particular, logic metaprogramming [36] uses a logic programming language to define metaprograms. The language being manipulated by the metaprogram need not be a declarative language [37]. When metaprogramming occurs across languages, the principle of stratification is naturally obeyed. The use of a declarative language avoids the subtle problems of class identity mentioned in section 2.2.1. It is possible to construct a declarative metaprogramming system that obeys the principles of encapsulation and correspondence, but neither property can be taken for granted.

6.3 Lisp

Historically, reflection was pioneered in Lisp, and the standard work on the semantics of reflection was done in the context of Lisp[11]. Object-oriented Lisp systems, as exemplified by CLOS, are the most germane to this paper.

Reflection in CLOS is supported via a Meta-Object Protocol (MOP) [18] that is part of the language definition. A MOP is a declarative model of the language ontology. The MOP is focused on support for reflection, including introspection, self-modification and, most notably, a rich notion of intercession.

CLOS meta-objects include (among others) classes. As in Smalltalk, classes are used both for application purposes, such as creating new instances (via the method make-instance) and maintaining shared state (via :class variables), and may have application specific methods as well. This contradicts the principle of stratification. The MOP largely upholds structural correspondence, but it only reifies entities that have run-time semantics.

6.4 Compile-time MOPs

Compile-time MOPs ([12], [13], [31]) have two key properties:

1. They deal with code: they only define meta-objects that reify entities that exist at compile-time.
2. They allow code to access the MOP while the code itself is being compiled. This lets the code influence how it will be compiled via compile-time computation, supporting a form of intercession. Code can even manipulate its own structure using the MOP, supporting generative programming ([14], [28]).

Item (1) implies that the meta-objects provided by a compile-time MOP are necessary but not sufficient to support the principle of correspondence. On the other hand, the ability to use these meta-objects in compile-time computation is not required by any of the design principles discussed in this paper. Further discussion of (2) is beyond the scope of this paper.

6.5 APT

APT (Annotation Processing Tool) [4] is a compile-time metaprogramming API designed to support the processing of metadata. The API is mirror based: it uses interfaces exclusively, and supports encapsulation and stratification. APT explicitly deals only with compile-time properties of the source language (Java), in line with the principle of correspondence. However, the API does not provide access to constructs below the method level. Unfortunately, APT is not integrated with a run-time reflection API.

6.6 C#/.Net

The C# reflection API supports introspection, as well as the dynamic creation and evaluation of programs, but not self-modification or intercession.

The API is mostly based on abstract classes. This allows alternate implementations to be derived by subclassing. However, the principle of encapsulation is not uniformly adhered to. In particular, the part of the API that supports the dynamic construction of programs does not use abstract classes or interfaces. It also appears that many of the abstract classes are not fully abstract, and thereby fix certain properties (especially representations) for all implementations. Despite these flaws, there appears to be considerable scope for alternate implementations, at least for introspection.

There is no clear-cut separation between the base-level and the meta-level. Classes directly support the reflective operations and the GetType operation is embedded in the root of the type hierarchy, object and cannot be overridden. Class types are also exposed via checked casts, the typeOf operator (the equivalent of Java's instanceOf) and hardwired notions of type identity.

While the API primarily reflects the .Net virtual machine, rather than the C# language itself, there is support for constructs like enumerations which appear to be in the domain of high-level languages. There is no distinct layer of the API dedicated to the high level language.

There does not appear to be a separation between code and computation. For example, methods support an Invoke operation that could not be supported when examining classes in a source code database.

6.7 Beta

The Beta [22] metaprogramming system Yggdrasil [24] automatically produces class hierarchies based on an abstract syntax given by a grammar, in close correspondence with the principle of structural correspondence. The generated hierarchies and associated tools support metaprogramming but not reflection. MetaBeta [10] provides support for run-time reflection including intercession. The distinction between Yggdrasil and MetaBeta is in line with the principle of temporal correspondence, but unfortunately the two APIs are unrelated.

6.8 Oberon

The reflective architecture of Oberon-2 [25] factors out reflection into a separate module, not unlike mirror-based systems. Reflective information is accessed through *riders*, iterator objects that support the traversal of the reified program. Riders are used for introspection of the program declarations and call stack and for dynamic execution. The system does not support self-modification or intercession.

Unlike mirrors, riders do not correspond directly to individual entities in a program. Instead, they represent sequences of similar entities. Riders correspond less directly to the language ontology, but appear to support stratification.

6.9 Firewall

Allen Wirfs-Brock et al. [34] discuss the properties of a declarative model for Smalltalk programs. The "abstract object model" they propose appears to be a mirror system for Smalltalk, implemented as the Firewall prototype for ParcPlace (now Cincom) Smalltalk. They discuss the advantages for distributed development and deployment. However, their discussion is Smalltalk specific and relies critically on the more general notion of a declarative program model for Smalltalk. They do not discuss the separation of high-level mirrors and low level ones, the interactions with static typing and multithreading or the relation with prototypes.

As Wirfs-Brock implies, a declarative language definition is a good basis for a clean mirror system. A key part of such a definition is the language's abstract syntax.

6.10 Aspect-Oriented Programming

Aspect-Oriented programming (AOP) has its roots in reflection and meta-object protocols in particular. One view of AOP is that it identifies a subset of reflective operations that are frequently useful for application development, and seeks to represent this subset at the base level via dedicated constructs. As such, AOP is deeply concerned with the distinction between meta-level and base-level operations. However, AOP relates only peripherally to this paper, where our chief concern is the design of meta-level APIs.

7. CONCLUSIONS

We have presented three design principles for meta-level facilities in object oriented programming languages:

1. **Encapsulation.** Meta-level facilities must encapsulate their implementation.
2. **Stratification.** Meta-level facilities must be separated from base-level functionality
3. **Ontological Correspondence.** The ontology of meta-level facilities should correspond to the ontology of the language they manipulate.

Mirror-based systems substantially embody these principles. They isolate an object-oriented programming language's reflective capabilities into separate intermediary objects called mirrors that directly correspond to language structures and make reflective code independent of a particular implementation.

As a result:

- Mirrors make remote/distributed development easier.
- Mirrors make deployment easier because reflection can be easily taken out or added, even dynamically.

The design principles behind mirrors may seem obvious, and yet these principles have not been widely applied to the reflective APIs of object-oriented programming languages.

Mirrors have been implemented in several different programming languages. These include class based languages, both dynamically and statically typed, as well as the prototype based language Self in which they were originally conceived. Mirrors have been successfully demonstrated in practice: very rich IDEs have been built using mirror-based reflection, as well as production quality debuggers.

The full power of mirror-based systems has yet to be realized. Systems that fully support metaprogramming of both code and computation at both the virtual machine and high-level language levels have yet to be demonstrated. However, the potential is clear.

Overall, we believe that the advantages of mirror-based systems greatly outweigh their disadvantages, and that mirror-based metaprogramming APIs should be the norm in future object-oriented languages.

8. ACKNOWLEDGMENTS

This work would not be possible without the teams who built the mirror-based systems described above.

The Self team: Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, Elgin Lee, John Maloney, Randy Smith, David Ungar and Mario Wolczko.

The Strongtalk team: Lars Bak, Gilad Bracha, Steffen Grarup, Robert Griesemer, David Griswold and Urs Hölzle.

The JDI team: Robert Field, Gordon Hirsch and James McIlroy.

The APT team: Joseph Darcy and Scott Seligman.

The authors are grateful to Christian Plesner Hansen and Kenneth Russell for productive discussions of these issues, and to Roel Wuyts, Stephane Ducasse, Mads Torgersen and Sophia Drossopolou and the SLURP group at Imperial College, as well as the anonymous referees, for helpful comments on earlier drafts of this paper.

9. REFERENCES

[1] Ole Agesen, Stephen N. Freund and John C. Mitchell. Adding Type Parameterization to the Java Language. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems Languages and Applications*, October 1997.

[2] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type inference of Self: Analysis of objects with dynamic and multiple inheritance. *Software - Practice & Experience*, 25(9):975-995, September 1995.

[3] Gul Agha. *Actors: A Model of Concurrent Computing in Distributed Systems*. MIT Press, Cambridge, Massachusetts, 1986.

[4] Annotation Processing Tool Home Page . http://java.sun.com/j2se/1.5.0/docs/guide/apt/

[5] Lars Bak, Gilad Bracha, Steffen Grarup, Robert Griesemer, David Griswold and Urs Hölzle. Mixins in Strongtalk. ECOOP Workshop on Inheritance, June 2002.

[6] Gilad Bracha. The Strongtalk Type System for Smalltalk, September 1996. OOPSLA Workshop on Extending the Smalltalk Language.

[7] Gilad Bracha and William Cook. Mixin-based Inheritance. In *Proceedings of the Joint ACM Conference on Object-Oriented Programming, Systems Languages and Applications and the European Conference on Object-Oriented Programming*, October 1990.

[8] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems Languages and Applications*, September 1993.

[9] Gilad Bracha and David Griswold. Extending Smalltalk with Mixins, September 1996. OOPSLA Workshop on Extending the Smalltalk Language.

[10] Soren Brandt and Rene Schmidt. Dynamic Reflection for a Statically Typed Language. Technical Report PB-505. Department of Computer Science, Aarhus University, June 1986

[11] Brian Cantwell Smith and Jim de Rivieres. Reflection and Semantics in LISP. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, 1984.

[12] Shigeru Chiba. A Metaobject Protocol for C++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1995.

[13] Shigeru Chiba. Macro Processing in Object-Oriented Languages. In *Proc. of Technology of Object-Oriented Languages and Systems* (TOOLS Pacific '98), Australia, November, IEEE Press, 1998.

[14] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming*. Addison-Wesley, Reading, Massachusetts, 2000.

[15] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.

[16] James Gosling, Bill Joy, Guy Steele and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, Reading, Massachusetts, 2004.

[17] Javadoc Tool Home Page. http://java.sun.com/j2se/javadoc/.

[18] Gregor Kiczales, Jim des Rivieres and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

[19] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems Languages and Applications*, October 1998.

[20] David. H. Lorenz and John Vlissides. Pluggable Reflection: Decoupling Meta-Interface and Implementation. In *Proceedings of the International Conference on Software Engineering*, May 2003

[21] Ole Lehrmann Madsen. *Keynote address*. OOPSLA, November 2002.

[22] Ole Lehrmann Madsen, Birger Moller-Pedersen and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, Reading, Massachusetts, 1993.

[23] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems Languages and Applications*, October 1987.

[24] Mjolner Informatics. The Mjolner System: Metaprogramming System. Available at http://www.mjolner.com/mjolner-system/yggdrasil_en.php

[25] Hans-Peter Mössenböck and Christoph Steindl. The Oberon-2 Reflection Model and its Applications. In *Proceedings of the Second International Conference on Metalevel Architectures and Reflection*, July 1999.

[26] Michael Richmond and James Noble. Reflections on Remote Reflection. Proceedings of the 24th Australasian conference on Computer science. Gold Coast, Queensland, Australia, pp.163 - 170, 2001.

[27] Self Programming Language Homepage. http://research.sun.com/research/self/

[28] Tim Sheard and Simon Peyton Jones. Template Meta-programming for Haskell. In *Haskell '02*. October, 2002. SIGPLAN Notices, 37, No. 12, pp. 60-75.

[29] Sun Microsystems. Java Platform Debugger Architecture. http://java.sun.com/products/jpda/.

[30] Sun Microsystems. A Metadata Facility for the Java^{TM} Programming Language. http://www.jcp.org/aboutJava/communityprocess/review/jsr175/

[31] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian and Kozo Itano. OpenJava: A Class-Based Macro System for Java. In *Reflection and Software Engineering*, LNCS 1826, Springer-Verlag, pp.117-133, 2000.

[32] David Ungar. Annotating Objects for Transport to Other Worlds. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems Languages and Applications*, October 1995.

[33] David Ungar and Randall Smith. SELF: The Power of Simplicity. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems Languages and Applications*, October 1987.

[34] Allen Wirfs-Brock, Juanita Ewing, Harold Williams and Brian Wilkerson. A Declarative Model for Defining Smalltalk Programs, October 1996. Invited talk at OOPSLA 96; available at http://www.smalltalksystems.com/_awss97/index.htm.

[35] Roel Wuyts. Declarative Reasoning about the Structure of Object-Oriented Systems, *Proceedings of TOOLS USA*, August 1998.

[36] Roel Wuyts, A Logic Meta - Programming Approach to Support the Co - Evolution of Object - Oriented Design and Implementation, Ph.D. thesis, Vrije Universiteit Brussel, 2001.

[37] Roel Wuyts and Stéphane Ducasse, Symbiotic Reflection between an Object - Oriented and a Logic Programming Language, in ECOOP 2001 International workshop on MultiParadigm Programming with Object - Oriented Languages, 2001.