Dynamic Proxies for Classes: Towards Type-Safe and Decoupled Remote Object Interaction

Patrick Thomas Eugster
Distributed Programming Laboratory
Swiss Federal Institute of Technology in Lausanne
CH-1015 Lausanne, Switzerland
patrick.eugster@epfl.ch

ABSTRACT

A dynamic proxy object is a typed proxy, created at runtime, conforming to a type specified by the application. Such an object can be used wherever an expression of the type it was created for is expected, yet reifies all invocations performed on it.

This simple but powerful concept has been introduced into Java at version 1.3 (and has later also appeared in the .NET platform). A dynamic proxy is created for a set of interfaces as an instance of a class, generated automatically on the fly without support from the Java compiler or virtual machine, implementing those interfaces. Unfortunately, dynamic proxies are only available "for interfaces". The case of creating dynamic proxies for a set of types including a class type, due to the increased complexity, has not been considered, meaning that it is currently not possible to create a dynamic proxy mimicking an instance of a class.

We present a pragmatic approach to supporting dynamic proxies "for classes", building on the existing solution to dynamic proxies for interfaces. We discuss the costs of such an extension, in terms of safety, security, and performance, and illustrate its usefulness through a novel abstraction for decoupled remote interaction, unifying (implicit future) remote method invocations and (type-based) publish/subscribe.

Keywords

Proxy, behavioral reflection, Java, remote method invocation, implicit future, publish/subscribe, peer-to-peer

1. INTRODUCTION

The benefits of the *proxy* design pattern and its relatives, such as the *decorator* pattern (responsibilities can be dynamically "attached" to objects) or the *adapter* pattern (method invocations performed on an expression can be "translated") [14] are well known in the field of object-oriented programming. A *dynamic* proxy is a typed proxy, created at run-time for a type (a set of types) defined by the application, which can be used in a consistent manner wherever an object of that type (of one of those types) is expected. An invocation performed on such a dynamic proxy object is however *reified*, somehow stepping from a statically typed context to dynamic interaction where *any* action can be performed in the confines of a method invocation. Together with dynamic invocation facilities, this concept enables graceful realizations of the above-mentioned patterns, and hence can be used to implement a form of *behavioral reflection* [22] (a.k.a. *computational reflection* [27]).

This concept of dynamic proxies has been added to Java^{TM}'s core reflection API [34] at version 1.3 as "library", that is, without specific support from the Java compiler or virtual machine [33]. A very similar mechanism has also appeared in Microsoft's .NET platform [36].

Since the introduction of dynamic proxies into Java, the WWW has whitnessed the appearance of a multitude of reports on using dynamic proxies to implement concepts such as *implicit* (*structural*) *conformance*, and more recently also *design by contract* [29] and *aspect-oriented programming* [23] (e.g. [11], [19] resp. for latter two). Dynamic proxies could further be used to implement *dynamic multi-dispatch*,[1] and last but not least, dynamic proxies leverage the proxy pattern also in the sense of the *remote method invocation* (RMI) paradigm. Thus, they represent an interesting alternative to the "standard", *static*, Java RMI proxies created explicitly through the `rmic` pre-compiler.

The implementation of dynamic proxies in Java is simple and elegant. When creating a dynamic proxy for an interface (type) `I`, an instance of a class implementing `I` is created, that class being generated automatically as byte code at run-time, loaded, and linked. Unfortunately however, the simplicity of this solution introduces important limitations. First, when used for behavior reflection, dynamic proxies only offer a limited support. Indeed, the common model [22] advocates the dynamic association of specific behavior through a *meta-level object* with a *base-level object*, emphasizing the transparency of this association provided to the base-level. When picturing a dynamic proxy as meta-level object, no support is given for managing the interaction between the proxy and its base-level object. Moreover, dynamic proxies suffer from the well-known weaknesses of proxy approaches; mainly the problems with "self" and encapsulation [25, 18]. The second and more important limitation of the current implementation of dynamic proxies in Java, which is also sensible in the context of behavioral reflection, defines the rationale of this paper. Dynamic proxies are namely only available "for interfaces", i.e., such objects can not be assigned to variables whose static type is a class (type). This might suffice for applying them in Java RMI (static types of variables referencing remotely invocable objects are always interfaces), which seems to have motivated

[1] See [9] for a presentation of dynamic multi-dispatch based on Java's dynamic invocation facilities.

their design. In general however, forcing developers to identify ahead all types for which dynamic proxies will be used and to introduce corresponding interfaces into type hierarchies at design, is strongly impeding.

The goal of this paper is not to present a drastically different and general approach to behavioral reflection for Java, as this has been successfully done by many authors in seminal work (e.g., [15, 10, 38], see Section 7). This paper presents a practical approach to augmenting Java's very model to support dynamic proxies "for classes",[2] in order to minimize the effort for programmers acuainted with the dynamic proxies as currently available. The proposed solution pursues the *extension* approach suggested already by the original implementation of dynamic proxies when creating a proxy class for a set of interfaces as *subtype* of those interfaces (in contrast to *envelopment* approaches relying on enclosing methods with hooks, e.g. [38, 10]). This leads to creating a proxy class for a class `C` as subclass of `C`. We present a set of byte code transformations for dealing with complications arising through this approach, including a general scheme for transforming instance field accesses to invocations of automatically created getter/setter methods (in order to be able to intercept such accesses), and schemes for dealing with methods and classes marked as `final`, and `private` methods and fields. The implementation of these transformations, which can be (nearly) independently enabled or disabled, requires neither a specific compiler nor an instrumented virtual machine, but rather relies on the established technique of performing byte code adaptation upon the loading of classes (cf. [1, 21, 30, 10, 38]). We present a modular algorithm for generating dynamic proxy classes for a set of types which can include a class, whose outcome, depending on the semantics of the input types and the enabling/disabling of the individual transformations, ranges from proxy classes with full functionalities over such with restricted possibilities, to nil. (For a set of types devoid of a class, the algorithm behaves like the original algorithm for dynamic proxies.) We discuss the "costs" of the presented transformation schemes, (1) in terms of *safety*, (2) *security*, and through measurements performed with the *SpecJVM* benchmark suite also in terms of (3) *performance*.

The particular motivation for this work is the implementation of a library-based type-safe abstraction for remote interaction in highly dynamic distributed settings (e.g., *peer-to-peer* environments). In such scenarios, strong decoupling of interacting components is desired, and dynamic proxies intuitively seem appealing to provide such decoupling. This abstraction, called *borrow/lend* (BL) (preliminary version presented in [12]), naturally unifies pass-by-reference (RMI) and pass-by-value ("messaging") interaction styles with decoupling flavors, i.e., *implicit future* RMI (cf. *wait-by-necessity* [6]) and *type-based publish/subscribe* ([13]). While benefits of (dynamic) proxies are long known in local settings, the use of patterns has in general only recently been put into the context of distributed programming [31]. We hence illustrate through the BL abstraction (and hence indirectly implicit futures and publish/subscribe) the decoupling capabilities of dynamic proxies. More precisely, we illustrate how they enforce decoupling in "distributed systems terms", i.e., in terms of *time* (components do not have to be up at the same time to initiate interaction), *space* (components do not depend on "references" to each other), *flow* (control threads of interacting components are not blocked), and *type* (interacting components are not required to use the exact same types). These decoupling flavors make of dynamic proxies, once extended to classes, a powerful mechanism for distributed programming exceeding the scope of simple RMI, which can be used to implement many abstractions for remote interaction in a type-safe way as "libraries".

[2]Though imprecise, this terminology is used for brevity.

Note that we by no means claim that Java's concept of dynamic proxies should be enhanced according to our approach. An officious goal of this paper is to illustrate through the case of dynamic proxies and the weaknesses of their realization, that future object-oriented programming languages, especially in the face of distributed programming, should be designed more uniformly, and with "full" reflection from the start.

The rest of the paper is organized as follows. Section 2 overviews the original concept and implementation of dynamic proxies in Java. Section 3 presents our modular approach to supporting dynamic proxies for classes, based on byte code transformations. Section 4 discusses implementation issues, including safety. Section 5 illustrates how we have used our extension to implement type-safe forms of asynchronous remote method invocations and publish/subscribe, through our BL abstraction. Section 6 discusses various issues, such as security impacts and limitations. Section 7 overviews related work. Section 8 concludes with final remarks.

2. BACKGROUND: DYNAMIC PROXIES IN JAVA

This section overviews the concept of dynamic proxies introduced with Java 1.3. This includes the types involved (Figure 1 overviews interaction between instances of these), the creation of dynamic proxy classes, and limitations.

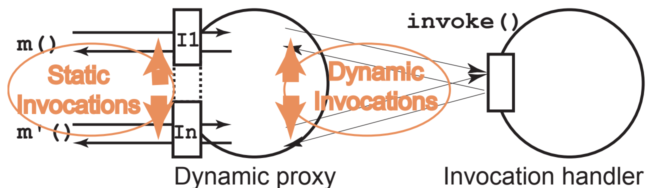


Figure 1: Interacting with dynamic proxies

2.1 Presentation

For presentation simplicity, we henceforth drop the qualifier "dynamic" when referring to proxies in the sense of Java reflection, unless confusion might otherwise arise (e.g., in contexts where *static* proxies generated by the RMI precompiler `rmic` appear). Furthermore, we omit the package `java.lang.reflect` common to all types for reflection except class meta-objects (instances of class `Class` in package `java.lang`).

2.2 Proxies

A proxy is an object which implements a non-empty set of interfaces {`I1`, ..., `In`}, for which that proxy ('s class)

was created through class `Proxy` depicted in Figure 2. That class extends class `Proxy` and implements all interfaces in the set {`I1`, ..., `In`} (see Section 2.4). Conforming to all the interfaces it was created for, a proxy can be cast to any of these types, and hence any method defined in such an interface can be invoked on it.

```
package java.lang.reflect.*;

import java.io.*;

public class Proxy implements Serializable {

  protected InvocationHandler h;

  protected Proxy(InvocationHandler h) { this.h = h; }

  public static
    InvocationHandler getInvocationHandler(Object proxy)
      throws IllegalArgumentException {...}
  public static Class getProxyClass(ClassLoader loader,
                                    Class[] interfaces)
      throws IllegalArgumentException {...}
  public static boolean isProxyClass(Class cl) {...}
  public static
    Object newProxyInstance(ClassLoader loader,
                            Class[] interfaces,
                            InvocationHandler h)
      throws IllegalArgumentException {...}
  ...
}
```

Figure 2: Class `Proxy` (excerpt)

2.3 Invocation Handlers

Every proxy has an associated invocation handler, which handles the method invocations performed on it. More precisely, invocations performed on a proxy object are reified and passed to its associated object of type `InvocationHandler` through latter object's `invoke()` method (see Figures 1,2). The arguments for an invocation of `invoke()` include (1) the object on which the method was originally invoked (i.e., the proxy), (2) a meta-object representing the method (an instance of class `Method`) that was originally invoked, and (3) the effective arguments (as an array of instances of the root object type `java.lang.Object`) for the invocation. The `invoke()` method is hence capable of handling any method invocation, which manifests in that the type of its return value and its arguments are of the root object type (values of primitives being transformed to the corresponding wrapper types), and it is declared to throw instances of `Throwable`.

According to the specification [33], an exception of type `NullPointerException` is thrown if `null` is returned by the `invoke()` method instead of a value of primitive type, and an exception of type `ClassCastException` is thrown if a wrong type is returned. An exception of type `UnknownThrowableException` can also be thrown upon invocation of a method `m()` on a proxy created for a set of interfaces {`I1`, ... `In`} in which at least two interfaces `Ii` and `Ij` declare `m()`, but with different sets of exceptions (see Section 6.3).

In essence, the `invoke()` method defined by invocation handlers can be seen as the symmetric counterpart to the `invoke()` method implemented by meta-objects representing methods. While the latter method allows to defer to run-time the choice of *which method to invoke*, the former method provides a means of deferring to run-time *what to perform upon method invocation*.

```
package java.lang.reflect;

public interface InvocationHandler {
  public Object invoke(Object proxy, Method method,
                       Object[] args) throws Throwable
}
```

Figure 3: The `InvocationHandler` interface

2.4 Creating Proxies

When invoking the `getProxyClass()` method in class `Proxy`, a set of interfaces, defined by their respective `Class` meta-objects, is specified. This leads to creating a proxy class, directly as byte code, for that set of interfaces (unless such a proxy class has already been created for that precise set) as a class which implements those interfaces. A further argument to the above method is a class loader, with which the possibly created class is to be loaded. In contrast to the `getProxyClass()` method, the `newProxyInstance()` method in addition instantiates the possibly generated proxy class. It hence takes an additional argument, which is an invocation handler, and does not return a reference to the proxy class, but rather an instance of that class which the specified invocation handler is associated with.

The `Proxy` class hence has a dual purpose. First, it serves as supertype for all proxy classes. Besides regrouping functionalities common to all proxy classes, this makes it possible to easily verify through the `instanceof` operator whether a given object is a proxy. Second, the `Proxy` class contains class methods, described above, which permit the generation of proxies/proxy classes, thus serving as "factory".

Note that in certain cases it is impossible to create a proxy class for a set of interfaces ([33]). Failures might especially occur whenever conflicts would also arise if a class was explicitly, i.e., statically, defined implementing the specified set of interfaces. Examples are the creation of a proxy class for two interfaces defining the same method with different return types (return types not being considered part of method signatures in Java), or for two non-public interfaces defined in two distinct packages. A proxy class for a set of interfaces including such with package visibility defined in the same package is created in that package, otherwise the package is unspecified. For the following, we suppose that package to be always the same, and simply denote it as `p`.

Supposing that none of the previously mentioned caveats apply, the generation of a proxy class for a given non-empty set of interfaces {`I1`, ... `In`} can be described through the algorithm presented in Figure 4.

Note that methods `equals()`, `hashCode()`, and `toString()` inherited by every class from `java.lang.Object` are handled just like custom methods. They are also overridden by proxy classes, and invocations to them are hence forwarded to the invocation handler of the respective proxy. Other methods defined in `java.lang.Object` are not overridden by proxy classes, as they are `final`.

create a proxy class I1...InProxy *extending* Proxy *implementing* I1, ..., In, *in package* p *or the package of the interface(s) with package visibility if exist(s), such that*

I1...InProxy *implements an empty private constructor*

I1...InProxy *implements a public constructor which expects an instance of* InvocationHandler *by passing that object to its superclass constructor*

for every interface I *in* {I1, ..., In}

for every instance method m() *declared in* I *or any of its superinterfaces (also recursively)*

I1...InProxy *implements* m() *by reifying and forwarding corresponding invocations to the* invoke() *method of the* InvocationHandler *associated with the proxy instance*

Figure 4: Algorithm for creating proxies

2.5 Illustration

Consider the interface I, defining a single method:

```
import java.io.*;

public interface I {
  public int foo(float f, String s) throws IOException;
}
```

The following lines illustrate the creation of a proxy for I. The invocation handler associated with the proxy simply prints the name of methods invoked on the proxy to the standard output, and then forwards the invocation to another object implementing I:

```
final I realI = ...;
InvocationHandler ih = new InvocationHandler() {
  public Object invoke(Object target, Method m,
                       Object[] args) throws Throwable {
    System.out.println("Method "+m.getName()+" invoked");
    return m.invoke(realI, args);
  }
};
I i = (I)Proxy.newProxyInstance(I.class.getClassLoader(),
                                new Class[]{I.class},ih);

i.foo(10.0, "hello");                > Method hello invoked
```

Figure 5 outlines pseudo code generated for the proxy class for I. For readability, we give source code and provide only a schematic view, which is of course also influenced by our own implementation dealing also with the additions described in the next sections. Furthermore, the class name is arbitrarily chosen: according to [33], the name space "$Proxy*" is reserved. Code fragments specific to custom interface I and its method(s) are written in *italics*. Details of the bodies of the overridden methods from Object are omitted in Figure 5 for simplicity.

2.6 Limitations

With respect to most work on behavioral reflection (see Section 7 in the context of Java), the present scheme is very basic. For associating a behavior through a proxy with an existing (base-level) object, the invocation handler associated with the corresponding proxy will most likely have to be given a reference to that object, and henceforth, the proxy has to be used instead of the original object, offering little transparency. More importantly, there is a problem with *self-invocations*: when a method of the effective target object invokes a further method of that very object, there is no way of intercepting that invocation (see also "self problem" [25]). Similarly, *self-references* returned by a base-level object invoked through a proxy would have to be recognized as such by the associated invocation handler, such that again a proxy could be returned instead (see also "encapsulation problem" [25]).

Yet, the detecting and handling of all such situations is very hard for two reasons. First, base-level objects can also return references to their fields which are difficult to identify as such, but would have to be shielded behind proxies as well. Second, and leading us to the main motivation for the present work, proxies can currently only be created for interfaces. Hence, creating a proxy for every value returned by a base-level object requires return types of all methods of such an object to be interfaces, and recursively also the return types of the methods of those objects, and so forth. When striving for a uniform model of behavioral reflection,

```
package p;

import java.lang.reflect.*;
import java.io.*;

public final class IProxy extends Proxy implements I {

  private static Method[] methods = new Method[4];

  static {
    methods[0] =
      Object.class.getDeclaredMethod("hashCode", null);
    methods[1] =
      Object.class.getDeclaredMethod("equals",
          new Class[]Object.class);
    methods[2] =
      Object.class.getDeclaredMethod("toString", null);
    methods[3] =
      I.class.getDeclaredMethod("foo",
          new Class[]{Float.TYPE, String.class});
  }

  private IProxy() {}
  public IProxy(InvocationHandler h) { super(h); }

  public int hashCode() {...}
  public boolean equals(Object o) {...}
  public String toString() {...}

  public int foo(float f, String s) throws IOException {
    try {
      return ((Integer)h.invoke(this, methods[3],
              new Object[]{new Float(f), s})).intValue();
    } catch(IOException e) { throw e; }
      catch(Error err) { throw err; }
      catch(RuntimeException rex) { throw rex; }
      catch(Throwable t) {
        throw new UndeclaredThrowableException(t);
    }
  }
  ...
}
```

Figure 5: A sample proxy class (schematic)

this would basically come down to programming exclusively with interfaces, and making use of classes only in instantiations. (Section 5 illustrates this inadequacy through a concrete case in the context of remote interaction.)

3. DYNAMIC PROXIES FOR CLASSES

This section proposes an enhancement of Java's current implementation of behavioral reflection, aiming at supporting the creation of proxies for classes.

3.1 Overview: Extension Approach

In essence, our approach builds on the principle applied for the generation of proxies for interfaces, that is, a proxy class for a set of types including a class is generated when needed at run-time as byte code, loaded, and linked. Since an instance of such a proxy class must conform to the class it was created for, and possibly an additional set of interfaces, the proxy class must not only implement those given interfaces, but must in addition subclass the class it is created for. This approach can be characterized as an *extension* approach since the classes to which reflection is to be applied are extended (i.e., subclassed), in contrast to many implementations of behavioral reflection which rather follow an *envelopment* approach (see Section 7). In order for a proxy to be able to reify any action performed on it, its class must hence override superclass members. Quite obviously, this works well in the case of non-private and non-final ("virtual") methods, but does not apply straightforwardly in many other cases. In the following, we propose a set of (nearly) independent transformations performed at byte code level upon class loading for dealing with those cases. Issues related to their implementation will be related by the next section, and to some extent in Section 6 regarding limitations.

Note that the choice of an extension approach has been motivated by the desire of retaining as much as possible of existing mechanisms and concepts of proxies for interfaces, but also by the fact that it adds no performance overhead in the, above-mentioned, most common case of invocations of a non-private and non-final ("virtual") methods (see Section 4.5).

3.2 Proxy Types and Access Handlers

As a direct consequence, proxy classes created for a set of types including a class cannot subclass class `Proxy`. As elucidated in Section 2.4, it is however very useful to have a common supertype for all proxy types, be it for the mere purpose of testing whether an object is indeed a proxy. To that end, we introduce the `ProxyType` interface (Figure 6). Furthermore, we introduce a type `AccessHandler` to reflect the possibility of performing (instance) field accesses in addition to (instance) method invocations in the case of proxies for classes. Field accesses made on a proxy created for a set of types including a class are handled namely through an instance of that `AccessHandler` type associated with the proxy. The `AccessHandler` type hence complements the `InvocationHandler` interface.

The `Proxy` class still serves as superclass for proxy classes created for a set of interfaces exclusively (see Figure 7), and hence implements `ProxyType`. This is depicted in Figure 8, in which additions in the new backwards-compatible version of the `Proxy` class are typed in *italics*.

Due to its dual purpose, the `Proxy` class has been

```
package java.lang.reflect;

import java.io.*;

public interface ProxyType extends Serializable {}

public interface AccessHandler {
  public Object get(Object proxy, Field field)
    throws RuntimeException;
  public void set(Object proxy, Field field, Object value)
    throws RuntimeException;
}

public class UnexpectedRuntimeException extends
    RuntimeException {
  private RuntimeException cause;
  public UnexpectedRuntimeException(RuntimeException re)
    { cause = re; }
  public RuntimeException getCause() { return cause; }
  ...
}
```

Figure 6: Types for dealing with field accesses

further slightly augmented, namely to capture the possibility of using it for creating proxy classes for classes. To that end, class `Proxy` has been added variants of the `getProxyClass()` and `newProxyInstance()` class methods enabling the lookup/generation of a proxy class for a (possibly empty) set of interfaces and a class, including instantiation of that proxy class in the second case.

Exceptions can be thrown, similarly to the original `Proxy` class, whenever the signatures of methods of supertypes for which the proxy class is to be created conflict, or visibility problems occur (see Section 2.4). On the other hand, the `get()` and `set()` methods in type `AccessHandler` are not supposed to generate exceptions (see Section 6.3).

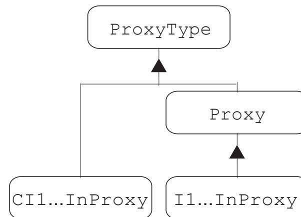


Figure 7: Proxy types

3.3 Handling Field Accesses

In order to support the reification of field accesses, these have to trigger method invocations. With an extension approach, a solution to this consists in replacing field accesses to invocations of getter/setter methods which are automatically generated for classes.

Since in Java, like in many other languages, fields can not be overridden by subclasses, but merely *hidden* [26], this can however not be achieved by straightforwardly defining a getter/setter method pair à la `getf()`/`setf()` for each field `f` defined by a class. Indeed, when dereferencing a variable to access a field, the search for that field starts at compilation from the static type of the variable, and proceeds along the superclass path, until the first corresponding field is found.

```
public class Proxy implements ProxyType {

  protected InvocationHandler h;

  protected Proxy(InvocationHandler h) { this.h = h; }

  public static
    InvocationHandler getInvocationHandler(Object proxy)
      throws IllegalArgumentException {...}
  public static boolean isProxyClass(Class cl) {...}
  public static Class getProxyClass(ClassLoader loader,
                                    Class[] interfaces)
      throws IllegalArgumentException {...}
  public static
    Object newProxyInstance(ClassLoader loader,
                            Class[] interfaces,
                            InvocationHandler h)
      throws IllegalArgumentException {...}

  public static
    AccessHandler getAccessHandler(Object proxy)
      throws IllegalArgumentException {...}
  public static Class getProxyClass(ClassLoader loader,
                                    Class cl,
                                    Class[] interfaces)
      throws IllegalArgumentException {...}
  public static
    Object newProxyInstance(ClassLoader loader, Class cl,
                            Class[] interfaces,
                            InvocationHandler ih,
                            AccessHandler fh)
      throws IllegalArgumentException {...}
  ...
}
```

Figure 8: Augmented `Proxy` class (excerpt)

This is opposed to the common case of (non-final instance) method invocations, and care must be taken that the information about the class in which an accessed field is declared is not lost.

A solution to this consists in conveying information about the declaring classes of fields by their respective getter/setter methods. This is illustrated in the following through three recursive subclasses:

```
class C1 {
  String f;
}
class C2 extends C1 {}
class C3 extends C2 {
  String f;
}
```

These classes are transformed to:

```
class C1 {
  String f;
  String get$C1$f() { return f; }
  void set$C1$f(String f) { this.f = f; }
}
class C2 extends C1 {}
class C3 extends C2 {
  String f;
  String get$C3$f() { return f; }
  void set$C3$f(String f) { this.f = f; }
}
```

Observe the corresponding transformations in code accessing these fields (source code for readability). The original code (left) and the code resulting from replacing those lines with corresponding transformations (right) in the original code ensure the same effect:

Original code:	Transformed code:

`C1 c1 = ...;`
`c1.f = ...;`
`... = c1.f;`
	`C1 c1 = ...`
	`c1.set$C1$f(...);`
	`... = c1.get$C1$f();`

`C2 c2 = ...;`
`c2.f = ...;`
`... = c2.f;`
	`C1 c1 =...`
	`c2.set$C1$f(...);`
	`... = c2.get$C1$f();`

`C3 c3 = ...;`
`c3.f = ...;`
`... = c3.f;`
	`C1 c1 =...`
	`c3.set$C3$f(...);`
	`... = c3.get$C3$f();`

This scheme ensures that always the right variable is accessed. It can be described as transforming a class `C` as follows:

$\mathbf{T}_{FA}$

for every access in `C` *to an instance field* `f` *decl. in class* `C'`
> *that access is made through the corresponding invocation of* `get$C'$f()/set$C'$f()`

for every instance field `f` *declared in* `C`
> *class* `C` *implements a getter/setter method pair* `get$C$f() /set$C$f()` *with corresponding signatures and the same modifiers as* `f`

Note that accesses to a field made in the access methods of that field are of course not transformed. Similarly, field accesses made in field initializations are retained.

Note furthermore that for simplicity, the package name of a class is supposed to be part of the class name in the above (and the following) algorithms. The names of getter/setter methods for a field declared in a given class namely contain the name of the class (in addition to that of the field, obviously), but also that of the package in which the class is contained (with occurrences of '.' replaced by '$'). Without this information, conflicts could occur in the case where a class called `C` in package `p2` subclasses another class called `C` in package `p1`, and both classes declare a same field.

3.4 Private Fields and Methods

The above modifications ensure that the behavior of a program is not altered, but unfortunately do not enable the reification of accesses to fields and methods which are marked as `private`. This stems from the fact that the dispatch of a private method does not start at the class of the invoked object, but rather at the class declaring the method (making use of the `invokespecial` rather than `invokevirtual` byte code operator [26]). Hence when a private method `m()` of a class `C1` is redefined in a subclass `C2` (e.g., a proxy class aiming at reifying method invocations), the method `m()` in `C2` will never be performed when one of `C2`'s instances is invoked through a variable who's static type is the superclass `C2` (as anyway the `m()` of `C2` could not be accessible in the same context in which `C1`'s `m()` was accessible).

To circumvent this bottleneck (not present in the original implementation of dynamic proxies, as interfaces only define public methods), getter/setter methods for private fields are defined with package visibility (the weakest visibility enabling overriding). The similarity between the search for private methods and the search for fields suggests the

adoption of a scheme for interception of application-defined private methods inspired by the one applied for field access transformations. The resulting scheme consists in complementing private methods with *stub methods*, through which former methods are invoked. Stub methods differ from the original methods in visibility (package visibility), and name (the name of the original method is prefixed with `C$`, `C` being the declaring class). Prefixes, just like infixes in getter/setter methods, are used to avoid accidental overriding in subclasses, since, as described above a class can very well declare a same private method as its superclass. Private methods are complemented by stub methods rather than modified directly because the renaming of `native` methods invalidates lookup tables of corresponding native libraries.

The schemes for dealing with private (1) fields and (2) methods can be described respectively as transforming a class `C` such that:

$\mathbf{T}_{PF}$

> *for every private instance field* `f` *declared in* `C`
>
> *the getter/setter methods* `get$C$f()/set$C$f()` *implemented by* `C` *are given package visibility*

$\mathbf{T}_{PM}$

> *for every private instance method* `m()` *declared in* `C`
>
> `C` *implements a stub method* `C$m()`*, with the same modifiers as* `m()` *but package visibility, by forwarding the invocation to* `m()`
>
> *for every invocation in* `C` *to a private instance method* `m()` *declared in* `C'`
>
> *that invocation is made through* `C'$m()`

It is important to note that, though a private method can only be invoked *inside* its declaring class `C`, it can not only be invoked by instances of `C` [17]. Indeed, certain inner classes can access members, including private methods, of their associated outer class instance.

For the following, $\mathbf{T}_{FA}$ will be considered as only dealing with fields declared with any visibility modifier other than private. As conveyed by the transformation above, $\mathbf{T}_{PF}$ nevertheless only makes sense if $\mathbf{T}_{FA}$ is enabled. This is however the only dependence that is observable in the transformations presented in this section.

3.5 Final Classes and Methods

As pointed out in [35], an extension approach works well as long as everything is "virtual", i.e., for classes which do not make use of the `final` keyword to rule out the possibility of subclassing them, or to prohibit the overriding of single methods.

There is no magic behind the solution to circumventing this limitation. It consists in handling final classes and methods as non-final ones when *linking* corresponding classes, yet keeping track of these occurrences for the *verification* of classes (see Section 4.1). When a non-proxy class is loaded violating final constraints, i.e., as subclass of a final class, or overriding a final method of its superclass, a loading error occurs, leading to an exception according to the specification of the Java virtual machine [26].

The scheme for dealing with (1) final classes and (2) final methods can be described respectively as transforming a class `C` such that:

$\mathbf{T}_{FC}$

> `C` *can be subclassed*

$\mathbf{T}_{FM}$

> *for every instance method* `m()` *declared in* `C`
>
> `m()` *can be overridden*

Hence, unlike in the original implementation of dynamic proxies, also methods defined in the root object type `Object` as final can now be overridden by proxies, even such created for interfaces only.

3.6 Superconstructor Calls

Remember that in Java every constructor (except the no-argument constructor in the root class `java.lang.Object`) must call a superclass constructor, whether this is explicitly coded as first instruction inside a subclass constructor, or a call to the default no-argument constructor of the superclass is automatically added [26].

At the same time however, a class `C` can explicitly define a private no-argument constructor, preventing this constructor to be used. This would also rule out the creation of a proxy class for `C`. One might immediately consider making such private no-argument constructors public in order to enable their invocation by default in subclass constructors. This approach however bears the danger of side-effects caused by such constructors (cf. [35]). To avoid such behavior, we create for each class `C` an *initializer class* `CInit` (a public class in package `p`, "empty" besides a public no-argument constructor), and a constructor taking an instance of `CInit`. Initializer classes are created such that if `C2` is a subclass of `C1`, `C2Init` is a subclass of `C1Init`. The constructor added to a class `C2` simply passes the instance of `C2Init` received as argument to the constructor of the superclass `C1`.

This scheme can be described as handling a class `C` as follows:

$\mathbf{T}_{SC}$

> *create a public class* `CInit` *in package* `p` *as subclass of* `C'Init` *where* `C'` *is the superclass of* `C`
>
> `C` *implements a public constructor which expects an instance of* `CInit` *by passing that instance to the corresponding constructor of* `C'`

3.7 Creating Proxies for Classes

With an extension approach, a proxy class generated for a class `C` is defined as subclass of that class `C` (and subtype of `ProxyType`), and overrides original instance methods of its superclass(es). Method bodies resemble those of class `IProxy` depicted in Figure 8.

To reify accesses to fields and invocations to private methods, a proxy class generated for a class also overrides all getter/setter and stub methods defined by its ancestor(s). Upon invocation, an access method constructs the reifications of the corresending field accesses and passes these to the `AccessHandler` associated with the proxy by calling the `get()` or `set()` method respectively.

Regarding the package in which a proxy class is created, the rule given in Section 2.4 applies without modifications. That is, a proxy class can only be created for a set of types including a class of which some types have package visibility

Transformation	Consequences for proxies if disabled
$\mathbf{T}_{SC}$	no proxies for classes with private no-argument constructors, otherwise possible side-effects
$\mathbf{T}_{FA}$	certain fields might not be initialized
$\mathbf{T}_{PM}$	private methods might behave arbitrarily
$\mathbf{T}_{FC}$	no proxies for final classes
$\mathbf{T}_{FM}$	final methods might behave arbitrarily

Table 1: Impact of transformations

if they are indeed defined in the same package. Furthermore, proxy creation can fail in similar situations that proxies for interfaces only (e.g., clashes in method declarations). In the case of a proxy class created for a class `C` and an interface `I`, both defining the same method yet with different visibilities (i.e., anything except public in the case of `C`), the proxy class implements that method as public.

The creation of a proxy class for a set of types including (possibly) a class `C` and a (possibly non-empty) set of interfaces {`I1`, ... `In`}, is described in the Figure 9. The algorithm is modular in the sense that it applies for any subset of the previously proposed transformations ($\mathbf{T}_{FA}$, $\mathbf{T}_{PF}$, $\mathbf{T}_{PM}$, $\mathbf{T}_{FC}$, $\mathbf{T}_{FM}$, and $\mathbf{T}_{SC}$) enabled, and in the absence of a specified class, behaves exactly like the algorithm presented in Section 2.4. When enabled, a transformation increases the scope of proxy creation. Inversely, when disabled, a transformation can lead to restricted use of the proxies created for certain classes, or even contradict proxy creation for safety reasons. When creating a proxy class for a class declaring fields without enabling $\mathbf{T}_{FA}$ for instance, care must be taken when using instances of that class, as fields might have not have been correctly instantiated. This is particularly valid if constructors generated by $\mathbf{T}_{SC}$ are used. Table 1 summarizes the consequences of disabling the different transformations. Note that $\mathbf{T}_{PF}$ is not reported, as it only makes sense when $\mathbf{T}_{FA}$ is enabled.

3.8 Illustration

Suppose a class `C`, implementing interface `I` introduced in Section 2.5, as follows:

```
public class C implements I {
  public String bar;
}
```

The following lines illustrate the creation of a proxy for `C` (the invocation handler `ih` from Section 2.5 is reused):

```
final C realC = new C();
AccessHandler ah = new AccessHandler() {
  public Object get(Object target, Field f) {
    System.out.println("Field "+f.getName()+" read");
    return f.get(realC);
  }
  public void set(Object target, Field f, Object val) {
    System.out.println("Field "+f.getName()+" written");
    f.set(realC, val);
  }
};
C c = (C)Proxy.newProxyInstance(C.class.getClassLoader(),
                                new Class[]{C.class},
                                null, ih, ah);

c.bar = "hello";                        > Field bar written
```

unless `C` *is final and* $\neg\mathbf{T}_{FC}$, *or* `C` *defines a private no-argument constructor and* $\neg\mathbf{T}_{SC}$

- *create a proxy class* `I1...InCProxy` *implementing* `ProxyType` *and* `I1`, ..., `In` *in package* `p`, *or the package of the type(s) with package visibility if exist(s), such that*
- `I1...InCProxy` *implements an empty private no-argument constructor*
- *if* `C`$=\perp$
  - `I1...InCProxy` *extends* `Proxy` *and implements a public constructor which expects an instance of* `InvocationHandler` *by passing that object to its superclass constructor*
- *else*
  - `I1...InCProxy` *extends* `C` *and implements a public constructor which expects each an instance of* `InvocationHandler` *and* `AccessHandler` *by retaining these, after*
  - *if* $\mathbf{T}_{SC}$
    - *creating an instance of* `CInit` *and passing it to the corresponding constructor of* `C`
  - *else*
    - *calling the public no-argument superclass constructor*
- *for every interface* `I` *in* {`I1`, ..., `In`}
  - *for every instance method* `m()` *declared in* `I` *or any of its superinterfaces (also recursively)*
    - `I1...InCProxy` *implements* `m()` *by reifing and forwarding the invocation to the* `invoke()` *method of the* `InvocationHandler` *associated with the proxy instance*
- *for every instance method* `m()` *originally declared in* `C` *or its superclass (also recursively) or any of its superinterfaces (also recursively)*
  - *if* `m()` *is not final, or* $\mathbf{T}_{FM}$
    - `I1...InCProxy` *implements* `m()`, *with public visibility if there is an interface* `Ij` *defining the same* `m()`, *by reifing and forwarding the invocation to the* `invoke()` *method of the* `InvocationHandler` *associated with the proxy instance*
    - *if* `m()` *is private and* $\mathbf{T}_{PM}$
      - `m()` *is renamed* `C$m()` *in* `I1...InCProxy`, *and given package visibility*
- *for every instance field* `f` *in* `C` *or its superclass (also recursively)*
  - `I1...InCProxy` *implements a getter/setter method pair* `get$C'$f()/set$C'$f()`, *where* `C'` *is the class defining* `f`, *by reifing and forwarding the invocation to the* `get()/set()` *methods of the* `AccessHandler` *associated with the proxy instance*
  - *if* `f` *is private and* $\mathbf{T}_{PF}$
    - `get$C'$f()/set$C'$f()` *are given package visibility in* `I1...InCProxy`

Figure 9: Full algorithm for creating proxies

The proxy class created upon the invocation of the `newProxyInstance()` method defined in the augmented `Proxy` class yields the pseudo code depicted in Figure 10. Code fragments typed in *italics* represent code depending on custom class `C` (fragments depending on interface `I` already highlighted in Figure 5 are not emphasized here).

4. IMPLEMENTATION ISSUES

This section discusses issues related to the implementation of dynamic proxies for classes.

4.1 Class Loading

Java's model for dynamic class loading and linking provides flexibility by letting the programmer "customize" class loading. However, this flexibility is limited for security reasons [16]. Custom class loaders defined by applications are intended to implement mainly algorithms for *finding* classes (acting as *initiating loaders* [24]), but are encouraged to delegate the effective *loading* of these classes to predefined, parent, loaders (acting as *defining loaders*), preferrably the default class loader itself. These specific algorithms are furthermore only to be applied if the default class loader itself fails in finding the class of interest.

Figure 11 overviews class loading and linking. The second class loader can delegate the loading of class `C` to its parent class loader, represented by the first class loader. Though originally pictured as component situated outside of the virtual machine, the default class loader can be viewed as part of the virtual machine. It acts as parent class for all user class loaders and has a direct link to the components for class verification and linking.

Many previous extensions to Java relying on byte code modifications, notably for reflection, have been implemented as user class loaders (cf. Section 7). Though implementable as user class loader, our transformations performed at byte code level have, similary to Agesen et al.'s proposal for genericity [1], been implemented through a preprocessor invoked during the virtual machine's default class loading and linking path.

This choice has been mainly motivated by safety reasons. Indeed, byte code *verification* must in our case apply before for instance the transformations required to intercept invocations of private methods, i.e., before *loading*. When (re-)implementing these verifications in a user class loader, it would have to be ensured that all classes are effectively loaded with that loader, which is often simply assumed in related approaches (especially older work relying on Java's outdated and less safe class loading scheme), yet can not be enforced. Table 2 overviews the dangers when implementing our modifications as user class loader.

4.2 Safety

The transformations described in the previous section introduce further dangers, which are however, unlike the ones summarized in Table 2, independent of the type of class loader the transformations are implemented with.

The semantics of an application can namely be altered when a class, accidentally or maliciously, overrides getter/setter methods or stub methods defined in its superclass. In particular, such overriding might jeopardize safety by giving access to otherwise hidden members.

Remember that to avoid that a class `C` defining the same (1) fields or (2) private methods as its superclass overrides

```
package p;

import java.lang.reflect.*;
import java.io.*;

public final class CIProxy extends C
    implements ProxyType, I
{

  private InvocationHandler ih;
  private AccessHandler ah;

  private static Method[] methods = new Method[12];
  private static Field[] fields = new Field[1];

  static {
    methods[0] =
      Object.class.getDeclaredMethod("getClass", null);
    methods[1] =
      Object.class.getDeclaredMethod("hashCode", null);
    ...
    methods[10] =
      Object.class.getDeclaredMethod("finalize", null);
    methods[11] =
      I.class.getDeclaredMethod("foo",
          new Class[]{Float.TYPE, String.class});
    fields[0] = C.class.getDeclaredField("bar");
  }

  private CIProxy() {}
  public CIProxy(InvocationHandler ih, AccessHandler ah)
  {
     super(new CInit());
     this.ih = ih;
     this.ah = ah;
  }

  public InvocationHandler getInvocationHandler()
    { return ih; }
  public AccessHandler getAccessHandler() { return ah; }

  public int foo(float f, String s) throws IOException {
    try {
      return ((Integer)ih.invoke(this, methods[0],
              new Object[]{new Float(f), s}).intValue();
    } catch(IOException e) { throw e; }
      catch(Error err) { throw err; }
      catch(RuntimeException rex) { throw rex; }
      catch(Throwable t) {
        throw new UndeclaredThrowableException(t);
    }
  }

  public String get$C$bar() {
    try {
      return (String)ah.invoke(this, fields[0]);
    } catch(RuntimeException rex) {
        throw new UnexpectedRuntimeExeption(rex);
    }
  }
  ...
}
```

Figure 10: A sample proxy class "for a class"

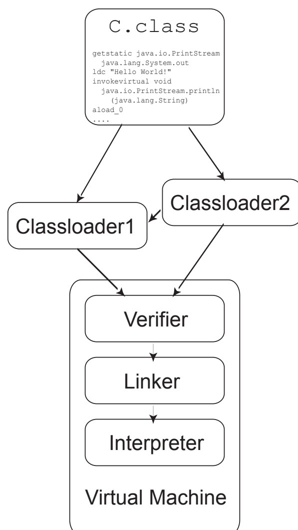


Figure 11: Class loading and delegation

Transformation	Dangers with user class loader
$\mathbf{T}_{FA}$	classes loaded with other loader can bypass field access interception
$\mathbf{T}_{PM}$	classes loaded with other loader can bypass private method invocation interception
$\mathbf{T}_{FC}$	classes loaded with other loader can subclass final classes
$\mathbf{T}_{FM}$	classes loaded with other loader can override final methods

Table 2: Dangers with user class loader

rather than hides these members, all methods (getter/setter methods in the first case, stub methods in the second case) added to `C` have been described so far as being "parameterized" by `C`. Indeed, the declaring class `C` of such members appears in the name of the added methods, and for similar reasons, the constructor added to `C` is similarly "parameterized" (indirectly), by defining it with a formal argument of a type whose name contains `C`.

This is a simplified representation. In fact, definitions added to a class `C` are instead parameterized by the result of a secret, though simple, hash function f(`C`). By proceeding this way, there is a vanishingly small probability that hidden definitions are overridden unintentionally, and also that hidden definitions can be overridden maliciously. Latter possibility would be provided if "fixed" names were chosen for added definitions (i.e., names containing directly `C`), even if a (constant) secret key was included. By observing one such name, e.g., through the printout of a stack trace upon the occurrence of an exception, specifically designed subclasses could be introduced easily into the system, compromising safety.

4.3 Introspection and Uniformity

Ensuring that introspection objects (e.g., instances of `Class`, `Method`) representing the structure of linked classes do not reflect changes made at load-time, further improves safety. Making these objects aware of reflection can also be exploited to promote uniformity, by making method invocations and field accesses made through introspection subject to reflection. As such, such instrumentations apply also to related approaches (where they are however not addressed).

`Class`: It must be ensured that `Class` is instantiated such that instances only return meta-objects representing "official" methods and constructors (others are shielded), and that originally final classes effectively appear as such.

`Method`: Similarly, instances of class `Method` have to reflect possibly removed `final` tags. Furthermore, to enable the handling of private methods invoked on proxies also through introspection, a corresponding instance of `Method` invoked dynamically through `invoke()` must forward such an invocation to the (shielded) instance of `Method` representing the corresponding stub method.

`Field`: When accessing the value of a field through introspection, e.g., through methods `get()` or `set()` of a `Field` meta-object, (or any methods for primitive types, e.g., `getfloat()`/`setfloat()`), the invocation must be transformed to an invocation of the (shielded) meta-object for the appropriate access method.

4.4 Lazy Resolving of Meta-Objects

As already pointed out, the illustrations provided for the creation of dynamic proxies in the case of a set of types consisting of only interfaces, but also in the case of a set including a class, are schematic. According to Figures 5 and 10 namely, all relevant meta-objects (representing methods and fields resp.) are resolved (looked up) at the creation of a proxy class. In practice, this static initialization is replaced by a lazy one. These static members are namely only resolved when first needed. For example, in class `CProxy` in Figure 10, the member `methods[11]` is only resolved when

the method `foo()` is effectively invoked on such a proxy for the first time.

4.5 Performance Considerations

Obviously, removing the effect of the `final` keyword affects performance by reducing the number of opportunities where the *just in time* (JIT) compiler can perform method inlining. However, JIT compilers such as Sun's HotSpot^{TM} virtual machine can also inline methods which are not final, and as long as no subclass overriding such an inlined method is loaded (e.g., a proxy class), the JIT compiler does not have to recompile an affected class. This occurs in our case only when a proxy class is loaded, and hence no sensible overhead was measured due to $\mathbf{T}_{FC}$ and $\mathbf{T}_{FM}$ in performance measurements. The main overhead associated with each of those transformations, and the only one in the case of $\mathbf{T}_{SC}$, becomes then the cost of performing the transformation itself, which is neglectable since classes are only loaded (and hence instrumented) once.

The transformations for field accesses ($\mathbf{T}_{FA}$) and private members ($\mathbf{T}_{PM}$ and $\mathbf{T}_{PF}$) are however more expensive in terms of overhead. Furthermore, it turns out that latter category has a stronger impact than former one. This is proof of a good programming discipline, as it reflects the level of encapsulation that is achieved with respect to fields. In any case however, the overheads are not as drastic as one might expect. These obvervations are conveyed by Figure 12, which elucidates results of performance measurements obtained with different sets of transformations enabled. These results were computed with the *SpecJVM* benchmark suite on a HP Omnibook XE$_3$, with a Pentium III processor, running Redhat Linux release 7.3. (Though enabled, $\mathbf{T}_{SC}$ is not mentioned in the figure, for the reason described above).

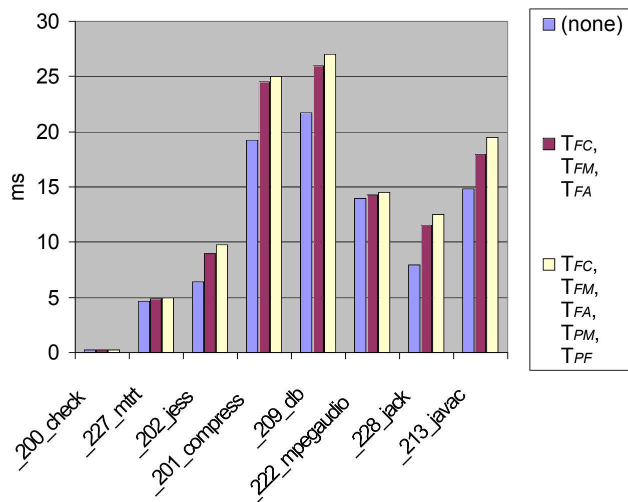


Figure 12: Performance overhead of transformations

Note that interface invocations are still slow (this is not a necessity, but a valid statement when considering current implementations of dynamic dispatch [2]), and that according to the original scheme for dynamic proxies these are always invoked through interfaces. If one would want to achieve a similar resilience with respect to behavioral reflection without our extension, i.e., one would like to be able to create a proxy for any object, one would hence not only have to introduce corresponding interfaces for all types in an application, but also deal with the overhead of interface invocations.

5. TYPE-SAFE AND DECOUPLED REMOTE OBJECT INTERACTION IN JAVA

In this section we illustrate how dynamic proxies for classes can be used in Java to support type-safe decoupled (cf. Section 1) remote object interaction, through the (implicit future) remote method invocation and (type-based) publish/subscribe paradigms. This illustration is made indirectly through the borrow/lend (BL) abstraction, which unifies these two paradigms. Quite obviously, not only those paradigms can benefit from dynamic proxies for classes, but also many other abstractions for remote interaction.

5.1 Borrowers and Lenders

With the BL abstraction (Figure 13), components can make objects, called *resources*, available to other components by indicating that they are willing to *lend* those objects. Conversely, components requiring resources can *borrow* such objects. Interaction between components hence takes place anonymously and indirectly, nevertheless explicitly (the goal is not to hide distribution), through resoures. A resource borrower can describe the resources it requires based on mainly three criteria, which must be met by a lent resource to make such interaction possible, namely:

Types: Borrower intererests are expressed for objects of a given type `B`, with which the type `L` of a resource borrowed from a lender "conforms". In fact, different "depths" of conformance between `L` and `B` are possible, ranging from explicit (name) conformance over different levels of implicit (structural) conformance to a form of completely dynamic interaction. More precisely, a depth of 0 represents explicit conformance (`L` has to be a declared subtype of `B` or `B` itself), while with a depth of 1 `L` only has to implement the members of `B`. With depth 2, types of fields and method parameters in `L` only have to be depth-1 conformant with those of `B`, etc. (see [4]). In order to enforce type safety, type constraints are expressed through a type parameter added to borrowers and lenders, which are namely implemented with Sun's compiler prototype for genericity [32].[3]

Names: Explicit names can be specified by borrowers and lenders, which must match for interaction to take place. Such names also play the role of symmetric keys, and are specified through a constructor argument for borrowers and lenders.

Predicates: Preferences can be expressed through predicates, based on the members of the type specified by borrowers and lenders. This will be more thoroughly illustrated throughout the rest of this section.

[3]The current solution relies on an extended compiler, another reason why we have never considered such an approach for creating proxies for classes.

```
package bl;

import java.io.*;
import java.rmi.*;

public abstract class Participant<R extends Resource>
    implements Serializable {
  public void activate()
    throws ActiveException, RemoteException {...}
  public void deactivate()
    throws InactiveException, RemoteException {...}
  public R constrain()
    throws InvalidConstraintException {...}
}

public final class Lender<R> extends Participant<R> {
  public Lender(R lent, String[] key) {...}
}

public final class Borrower<R> extends Participant<R> {
  public Borrower(Inbox<R> in, String[] key) {...}
}

public interface Inbox<R> {
  public void deliver(R r);
}
```

Figure 13: Borrowers and lenders (excerpt)

5.2 Resources

Resources are the key in the BL abstraction to unifying different remote interaction styles, and are split in two main categories. When resources manifest (1) *pass-by-reference semantics*, the BL abstraction plays the role of lookup service for implicit future remote method invocations (cf. [6]). More precisely, such a (coarse grain) resource remains on its lender's site, and proxies are provided to borrowers, through which they can interact with the resource. Conversely, through resources with (2) *pass-by-value semantics*, the BL abstraction implements a type-based publish/subscribe (TPS) interaction model [13]. Such resources represent namely (fine grain) event objects, of which copies are created for each matching borrower.

The BL abstraction encompasses many resource subtypes (see Figure 14). For instance, "lazy pass-by-value" resources exphibit both pass-by-value *and* pass-by-reference flavors (`DownloadResource`). They can be transferred by value, on demand, or automatically upon their first invocation. Further types of resources include resources with support for dynamic interaction (`DynamicResource`), which are not required to provide statically defined interfaces, replicated resources (`ReplicatedResource`), or "replaceable" resources (`ReplaceableResource`). Combinations of these abstract resource types are also possible.

No matter the type of semantics however, matching resources are in any case "delivered" to borrowers as proxies (`deliver()` in `Inbox`). This promotes decoupling, as will be illustrated in Section 5.5.

5.3 Contract Methods

Resource types have individual predefined methods reflecting the "contracts" introduced by the use of such resources. These *contract methods* vary strongly in semantics. Some can return specific values, while others *can be* implemented by a resource class, but *do not have to be*, i.e., they

```
package bl;

import java.io.*;
import java.rmi.*;

public interface Resource {
  public void setConformance(int depth);
  public void setProtocol(Protocol p)
    throws NoSupportException;
  public void setQoS(QoS qos) throws NoSupportException;
}

public interface ValueResource
    extends Resource, Serializable {}

public interface ReferenceResource
    extends Resource, Remote
{
  public void setSynchronization(boolean noLazy)
}

public interface DownloadResource<R extends ValueResource>
    extends RemoteResource
{
  public void setDownload(boolean automatic)
  public R download(Protocol p)
    throws RemoteException, NoSupportException;
}

public interface DynamicResource extends Resource {
  public DynamicResource invoke(String methodName,
                                Object args[])
    throws InvalidMemberException, NoSupportException;
  public DynamicResource get(String fieldName)
    throws InvalidMemberException, NoSupportException;
  public ResourceDescription getDescription();
}

public interface ReplicatedResource extends RemoteResource
{
  public void joinGroup() throws NoSupportException;
  public void leaveGroup() throws NoSupportException;
  byte[] getState() throws NoSupportException;
  void setState(byte[] b) throws NoSupportException;
}

public interface ReplaceableResource<R extends Resource>
    extends Resource
{
  public void replace(R new) throws RemoteException;
}
```

Figure 14: Basic resource types (excerpt)

can have empty bodies. As an example, the basic resource type `Resource` contains methods allowing the setting of QoS parameters (borrowers) or the transmission protocols to be used (borrowers and lenders). These methods are used when expressing preferences through the `constrain()` method of a participant. When invoked, that method in fact returns a dynamic proxy, which "registers" the invocations subsequently performed on it.

5.4 Illustration

The above concepts, mostly predicates, are best illustrated by an example. Consider the scenario of songs being shared throughout the Internet. A typical Java type for incarnating such songs could look like the following:

```
import java.rmi.*;
import javax.sound.sampled.*;
import bl.*;

public class Song implements DownloadResource<Song> {
  public String getTitle() throws RemoteException {...}
  public String getArtist() throws RemoteException {...}
  public String getGenre() throws RemoteException {...}
  public AudioInputStream getStream()
    throws RemoteException {...}
  public Song download(Protocol p)
    throws RemoteException, NoSupportException {...}
  public Song(String title, ...)
    throws RemoteException {...}
}
```

An instance of `Song` hence conveys information about (1) the title, (2) the artist, (3) a genre description, and (4) the effective track (type `AudioInputStream` in package `javax.sound.sampled`) for a song. The `download()` contract method does not have to be implemented (it can simply return `null`).

A song produced by a record company eReC could then be shared with (of course paying) customers as follows (exception handling omitted for simplicity, invocations made on proxies typed in *italics*):

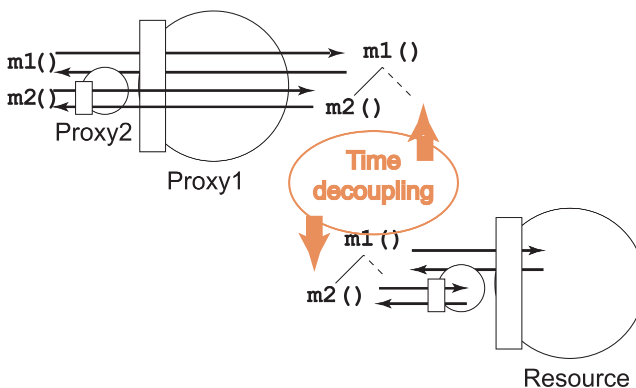
```
Song lSong = new Song("The next love song",
                      "The next boys band", ...);
String key = ...;
Lender<Song> sLender =
  new Lender<Song>(lSong, new String[]{"eReC", key});
sLender.setProtocol(new ftp(...));
sLender.activate();
...
```

Symmetrically, interest in songs can then be expressed by customers like the following:

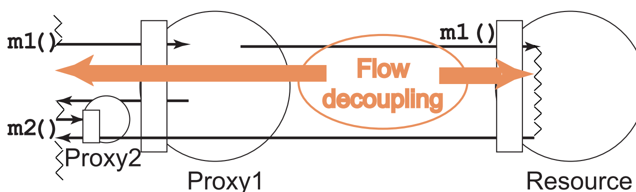
```
String key = ...;
Borrower<Song> songs =
  new Borrower<Song>(new Inbox<Song>() {
    public void deliver(Song bSong) {
      if (!Jukebox.isQueued(bSong));
      Song localS = bSong.download(new ftp(...));
      Jukebox.queue(localS);
    }
  }, new String[]{"eReC", key});
Song pSong = songs.constrain();
pSong.setConformance(0);
pSong.getArtist().equals("The next boys band");
songs.activate();
...
```

Here, duplicates are filtered by storing songs in a jukebox and checking whether a (new) received song is already present. To that end, songs are compared through `getTitle()` and `getArtist()` invoked remotely on the new song represented through `bSong` by the jukebox (details omitted). Only if not already present, the song is then downloaded by invoking the `download()` method like a normal remote method with lazy synchronization (synchronous invocations and automatic download are by default disabled). Thanks to the lazy synchronization of that download procedure, the song can be "put" in the queue of the jukebox before having effectively been entirely received at that point, and the corresponding thread can process the next new song.

This example illustrates predicates, including such based on contract methods. Above, interests are expressed only in objects which explicitly conform to the `Song` type *and* represent songs given by a certain band (a logical *or* of several conditions requires these to be expressed on individual proxies obtained by successive calls to `constrain()`).



(a) Predicate expression



(b) Lazy synchronization

Figure 15: Dynamic proxies in the BL abstraction

5.5 Proxies and Decoupling

Through its broad use of proxies, the BL abstraction achieves strong "decoupling" of interacting components. While the decoupling flavor of (dynamic) proxies is well known and understood in local settings, we illustrate it in the following in "distributed systems terms", by considering different aspects of decoupling (see Figures 15(a),15(b),16(a), and16(b); invocation handlers have been omitted for simplicity).

The important value added by the implementation of

proxies for classes to all these forms of decoupling consists in providing flexibility and static safety for the design of resource types. Indeed, without the presented extension, types of borrowed and lent resources would obviously all have to be defined as interfaces (unlike `Talk`), but more importantly, also return types of their methods (recursively). This constraint is particularly annoying as it *can not be ensured at compile-time.*

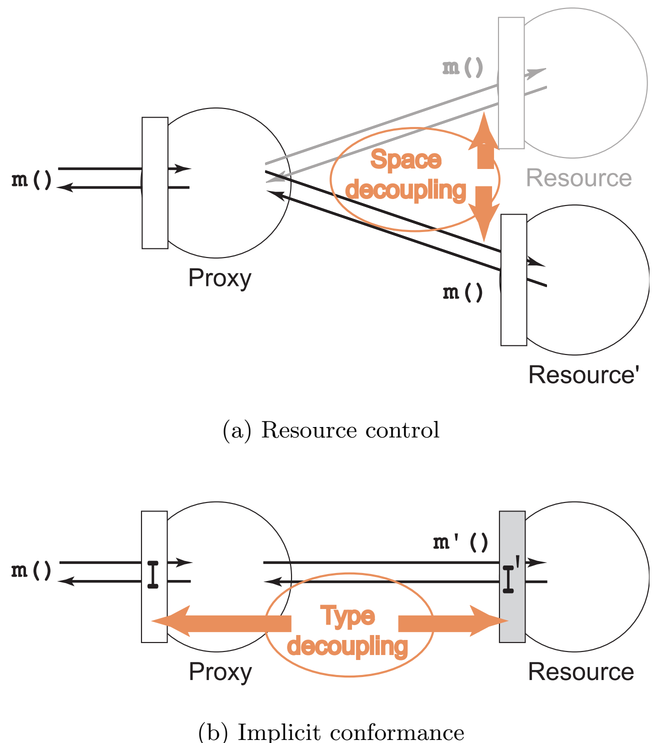
Time decoupling: When *expressing predicates*, i.e., "registering" contract method invocations as well as invocations (and field accesses) performed in the context of resource-specific predicates with proxies, time decoupling is achieved between components. A borrower can obtain access to resources whose lenders were not even running at the time the borrower expressed its predicates (Figure 15(a)). In the example above, the borrower can receive new songs of "The next boys band" produced by eReC *after* the borrower expressed its interests through `pSong`. The decorator pattern suggested by expressing preferences through contract methods has a further considerabl benefit: in contrast to the regrouping of contract methods in abstract resource classes to be subclassed by application-defined resource classes, resource class hierarchies are not polluted. This is a substantial benefit in a language with single inheritance such as Java.

Flow decoupling: By delivering resources as proxies, *lazy synchronization* can be provided when invoking remote resources (implicit future RMI, Figure 15(b)), but also when automatically transferring lazy pass-by-value resources upon invocation. This provides flow decoupling of components in all scenarios. In the above example, a song can be queued for playing before it has been entirely downloaded.

Space decoupling: Space decoupling is nicely demonstrated through *resource control*. When a lender invalidates a lent resource (e.g., `sLender.deactivate()` in the above example) or replaces it by a new one (e.g., by invoking `replace()` on the lender side if `Song` had been defined as subtype of `ReplaceableResource`), a borrower's reference to (the local copy of) that resource can be updated if it is a dynamic proxy (Figure 16(a)). Without hooks into the virtual machine, this is in fact the only way of transparently to the programmer swapping at any moment the object pointed to by a variable of arbitrary type.

Type decoupling: The adapter pattern supported by dynamic proxies can be used to implement *implicit conformance* between types of lent and borrowed resources (Figure 16(b)), leading to type decoupling. If the borrower in the above example expressed interest in a type `Song2` not related to `Song` but encompassing a subset of the members of `Song`, it could nevertheless receive the published song by adapting its predicates (e.g., `pSong.setConformance(1)`).

We believe that with the same combination of genericity and dynamic proxies for classes, further abstractions for distributed programming, such as *tuple spaces* [28] or even *join patterns* [5] could be implemented without specific extensions to the Java language, in a type-safe manner.



(a) Resource control

(b) Implicit conformance

Figure 16: Dynamic proxies in the BL abstraction

6. DISCUSSION

This section discusses various issues, such as security implications and limitations (beyond limitations of the proxy model for behavioral reflection already discussed) tied to proxies. In the face of these issues, we compare the original implementation of dynamic proxies, and our augmented implementation.

6.1 Security

No programming language has so far undergone as intensive investigations in terms of security as Java.[4]

Core notions in Java security are *protection domains* and *permissions* [16]. Protection domains correspond to certificates for signing classes, and/or URLs for obtaining classes, and have an associated set of permissions. (If a class maliciously exploits the permissions associated with its protection domain, the *principals* associated with that domain can be held responsible.) Java system classes are part of a system protection domain which by default includes all permissions. A security *policy* in Java governs the permissions granted to the different protection domains.

Two key concepts underlying security in Java are (1) the principle of least privilege and (2) the concept of permission intersection. The former principle states that a piece of code should operate with *the smallest possible set* of privileges. The latter concept requires that, when performing a piece of code, *the entire set* of protection domains represented through classes on the execution stack at that point include

[4]An in-depth presentation of security issues raised by general-purpose reflective extensions to Java can be found in [8].

the permissions necessary for executing that code.

In the context of dynamic proxies, handlers are the central players with respect to security. The (augmented) `Proxy` class, as well as created proxy classes are namely, just like any system classes, given all permissions. When an object accesses another object via a proxy, the only relevant classes added to the execution stack are thus handler classes. Besides a class implementing `InvocationHandler`, this potentially includes a class implementing `AccessHandler` in the case of proxies for classes. On the one hand, care must be thus taken when inserting a proxy between a caller and a callee, to not make the interaction impossible by associating an instance of a handler class with an insufficient set of permissions with that proxy. On the other hand, one can exploit this to dynamically introduce security barriers. Rather than providing third parties with direct access to a class, these are urged to access that class through a dynamic proxy. Handlers can then at run-time decide on granting permissions or not.

6.2 Resolved Meta-Objects

Suppose the invocation of a method `m()` defined in two distinct interfaces `I1` and `I2` on a proxy implementing both those interfaces. With the original implementation of dynamic proxies in Java, the method meta-object resolved upon that invocation depends on the order in which `I1` and `I2` are specified upon creation of the proxy (class), rather than on the static type of the variable through which the proxy was invoked. In other terms, though invoked through a variable of static type `I2`, the method meta-object transmitted to the invocation handler bound to the invoked proxy represents `m()` in `I1`, should the interfaces have been specified in the order 1. `I1`, 2. `I2` when creating that proxy. The reason for this is that the information of the static type of such a variable is lost through the dynamic dispatch undergone by the invocation.

For the same reason, the meta-object resolved when invoking a method on a proxy created for class `C` and a set of interfaces can not be used to deduce the static type of the variable through which the proxy was invoked. By convention, the class `C` takes precedence over the interfaces orderwise in the above situation.

More useful information is however obtained upon field accesses with our extension. Since the information of the declaring class of an accessed field is contained in the name of the used getter/setter method, the resolved meta-object in the case of a field access reification can reflect faithfully the class containing the accessed field (which is however not necessarily the static type of the variable through which the field was accessed).

6.3 Exceptions

With the impossibility, described above, of determining the static type of a variable through which a proxy has been invoked, restrictions occur also in the original implementation of dynamic proxies with respect to exceptions thrown by methods. In the above example of two interfaces `I1` and `I2` defining a same method `m()`, both definitions can very well encompass different sets of exceptions that are declared to be thrown by `m()`. Hence, the implementation of `m()` in a proxy class can only throw exceptions in the intersection of *all* sets of exceptions declared to be thrown by the different declarations of `m()`. Any other exception returned by the invocation associated with a proxy will be wrapped by an instance of `UndeclaredThrowableException`, as depicted by Figure 5.

In the context of proxies for classes, the introduction of getter/setter methods to replace direct field accesses does not per se add new possibilities for the raising of exceptions. However, when invoking such a proxy, the associated field access handler can very well return an object of a wrong type, leading to a `ClassCastException`, or generate other types of `RumtimeException`s. Since according to the Java language specification [17] a field access is not expected to do so, such exceptions are wrapped by the `UnexpectedRuntimeException` type, as shown in Figure 10.

6.4 Primitive Types

A remaining limitation related to dynamic proxies appears through the BL abstraction outlined in the previous section, where predicate expression, just like implicit future invocations, can not conclude when a primitive type is encountered as return type of a method. For instance, changing the second predicate in the example to the following would not be possible:

```
pSong.getArtist().equals("The next boys band") == false;
```

This can be circumvented, just like the operator overloading lacking in Java, by introducing a package containing own wrapper classes for primitive types. The main differences between these classes and the original wrapper classes would be the addition of (1) a method `isEqual()` returning a value of (the alternative) wrapper type `Boolean` rather than the primitive type `boolean`, and also (2) methods such as `plus()` and `minus()` reflecting operators. The alternative class `Boolean` also has to be equipped with methods such as `and()`, `or()`, `not()`. Introducing own wrapper classes can also help preventing the modification of Java system classes such as standard wrapper classes, which might, should they be passed further and exploited, infringe license terms.

The fact that first-class constructs like exceptions or `if...else` statements can not be taken into account when expressing borrower preferences either, is however not seen as a drawback, but more as a welcome limitation of predicate semantics. As we have shown in [13], an efficient implementation of distributed filtering and routing based on predicates (and hence methods) would namely become hard if not impossible [13] without restricted semantics for predicate code.

6.5 Static Methods and Fields

Throughout the previous chapters, we have tackled the issues of handling *instance* field and method invocations, without addressing class (`static`) methods and fields. In fact, class methods and fields do not seem to fit naturally into a proxy model, as proxies are considered as objects "wrapping" other objects, i.e., instances.

This in fact illustrates a mismatch between the common definition of class members and distributed contexts. Indeed such members represent class-wide features, but *with respect to a single process* (e.g., a virtual machine) only. The terminology (e.g., "class method") as well as the notation used to access such class members (e.g., `C.m()` where `C` is a class name) are hence not suited for distributed contexts, as they hide the above-mentioned, indeed important, restriction. While extending the scope of class declarations

to *all processes* would become prohibitively expensive in a distributed setting, it could be very interesting to be able to access a class member in another process, for instance through a proxy to an object in that process.

7. RELATED WORK

The subject of reflection has benefitted from much research effort in the context of Java. We focus on efforts around behavioral reflection, involving the use of specific classloaders, or being applied to distributed contexts.[5]

7.1 Kava

Kava [38] is a general extension to Java reflection providing behavioral reflection, relying on a specific user class loader to modify classes at load-time. Kava however follows an envelopment approach (unlike its proxy-based predecessor Dalang [37]), in the sense that hooks are added *around method invocations* and field accesses, to pass control to the meta-level.

In the context of dynamic proxies for classes, such an approach could be adapted to tranform the lines (source code for readability)

```
C c = ...;
String f = c.f;
```

into something looking like the following (by omitting exceptions etc.):

```
C c = ...;
String f;
  if (c instanceof ProxyType)
    try {
      Field F = C.class.getField("f"));
      f = (String) Proxy.getAccessHandler(c).get(c, F);
    } catch(Exception ex) {}
  else f = c.f;
```

Such an approach enables the seemingly uniform interception of any method invocations and field accesses, including class methods and fields. This however comes at the expense of a sensible overhead through the use of introspection for *every* field access or invocation on through proxy. In contrast, with our approach, such expensive calls to the core reflection API are made *at most once* for a same method/field for all uses of that member (see 4.4).

Since an instrumented *user* class loader can be bypassed (see Section 4.1), the uniform application of reflection is however put at stake [37]. Just like in our case, hooks affect classes reflected upon (e.g., for invocations to own instances) as well as classes using former classes (e.g., invoking instances of classes reflected upon). The interception of method invocations and field accesses made through introspection classes is not discussed in [38], further reducing uniformity.

7.2 Javassist

Javassist [10] is another extension to Java reflection, promoting load-time *structural* reflection. Javassist offers a low-level API operating at byte code level, and a more high-level API providing useful "macros" built on former one (e.g., addition, modification of methods), including a specific classloader for the instrumentation of classes.

[5][38] and [8] present more detailed surveys of existing reflective extensions to Java.

Javassist is extremely general and powerful, and has many potential applications. Behavioral reflection is in fact only of these instantiations, obtained by *wrapping methods*. I.e., shifts to the meta-level are achieved by inserting hooks *into* the bodies of methods to be reflected upon rather than *around* the invocations to them (as in Kava). This scheme, in contrast to Kava, establishes a clear equivalence between the classes reflected upon and the classes that have to be modified, i.e., loaded with Javassist's specific class loader. Since this scheme can not be extended to field accesses, Javassist proceeds similarly to our approach in that case by replacing field accesses by invocations, however to class methods.

The general applicability of Javassist has been illustrated by realizing *binary code adaptation* [21], aspect-oriented programming [20], or a form of synchronous RMI without static proxy generation. Based on the latter experience, Addistant, another instantiation of Javassist, is described in [35]. Addistant aims at the distribution of "legacy" Java programs, that is, Java programs developed without distribution in mind. This a posteriori distribution is discussed from a language perspective, leaving aside the handling of failure patterns introduced by the now distributed nature of applications.

Four different ways of modifying a class to reflect the possibly remote location of certain of its instances are discussed. In the case of a class whose instances are *all* remote, the class can for instance be *replaced* by a proxy class. An extension approach (termed *subclass approach* in [35]) is also discussed. The problems with final classes and methods are pointed out, unlike the cases of private methods and field accesses.

Last but not least, Javassist has been used to implement a very first prototype of the extension presented in this paper.

7.3 ProActive

ProActive, a descendant of Java// ("Java parallel"), is similar to Addistant, in that it aims at providing features for "transparent" distributed or parallel execution of Java programs [7]. ProActive advocates the use of implicit futures to decouple remotely interacting components, where proxies are obtained at run-time by manually instantiating proxy classes part of the ProActive libraries. Implementation details are not provided, but it seems that our extension (even along with genericity as presented in Section 5) could be applied to provide transparency and type safety in ProActive as well.

8. CONCLUSIONS

While several authors have suggested ways of augmenting Java's reflection capabilities in the large, this paper presented an approach to broadening the scope of Java's own concept of dynamic proxies, in order to make it available also for classes.

The solution presented in this paper makes neither use of a specific compiler nor of an instrumented virtual machine, but can do with a set of manipulations performed at class loading. For instance, to be able to intercept field accesses we have presented a scheme for transforming such accesses to invocations of automatically generated getter/setter methods; a general transformation scheme whose applicability is not limited to the generation of dynamic proxies and the Java language.

By performing the proposed modifications by preprocessing classes upon loading, as pointed out in this paper, a nearly uniform model of behavioral reflection can be achieved. This includes "delicate" cases such as final classes and methods, and private methods and fields, and makes a case against the claim that the achievement of field access interception with a proxy approach is impossible without specifically modified virtual machine [8]. Furthermore, we have considered performance, safety, and security implications, and proposed how to adapt introspection classes to unify the behavior of method invocation and field access interceptions made through introspection and those made statically.

The remaining limitations of our extension are due to Java itself. The original model for dynamic proxies for interfaces suggests a proxy object (a base-level object) and an effective target object (its associated meta-level object) to appear with distinct identities. While the present approach to dynamic proxies for classes could be adapted to collapse these two identities (as suggested by [18] for instance), the main remaining drawback seems to be, for our applications in decoupled distributed programming illustrated in this paper, the fact that Java is a hybrid language. Indeed, primitive types are difficult to integrate into a uniform model of reflection. Hence, it would be interesting to see how one could combine our approach with a uniform object model such as the one promoted by Kava [3].[6]

We believe that the issues addressed in this paper are not of relevance only for the BL abstraction and Java. As pointed out, BL unifies implicit future RMI and type-based publish/subscribe, two paradigms which could obviously benefit in isolation from the proposed extensions. Furthermore, Microsoft's .NET platform [36], inspired by Java, proposes a closely related concept of dynamic proxies, with nearly the same limitations. For instance, field accesses can not be intercepted either, which is however counterbalanced by the fact that types in .NET languages such as C# can declare *properties*, a form of fields with inherent support for getter/setter methods.

The question then remains why one would use public fields at all, as at least parts of the object-oriented community agree that the shielding of fields (i.e., encapsulation) should not only be a design choice. Otherwise, one could argue for a *remote field access* (RFA) paradigm parallel to RMI. One of the main reasons why the RMI has become so popular is namely precisely the impression of uniformity it provides (and which is often misinterpreted). That is, interacting with remote objects looks very much like interacting with local ones. Strangely enough however, only little effort has been made to improve uniformity through a RFA facility (we know only of [35]). Especially in the face of modern abstractions for remote interaction, which tend to make also code fragments mobile by executing them remotely, it becomes hard to clearly separate local and remote definitions, and uniformity should then become an ever more important issue. Similar questions can be posed in the context of `static` definitions, whose semantics might be worth revisiting in a distributed setting

9. REFERENCES

[6] Not to be confused with the Kava approach to behavioral reflection in Java [38], cf. Section 7.1.

[1] O. Agesen, S.N. Freund, and J.C. Mitchell. Adding Type Parameterization to the Java Language. In *Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97)*, pages 49–65, October 1997.

[2] B. Alpern, A. Cocchi, S. Fink, and D. Grove. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. In *Proceedings of the 16th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001)*, pages 108–124, October 2001.

[3] D. Bacon. Kava: A Java Dialect with a Uniform Object Model for Lightweight Classes. In *Proceedings of the Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 68–77, June 2001.

[4] S. Baehni, P.Th. Eugster, and R. Guerraoui. OS Support for Peer-to-Peer Programming. In *Proceedings of the 22th IEEE International Conference on Distributed Computing Systems (ICDCS '02)*, pages 355–362, July 2002.

[5] N. Benton, L. Cardelli, and C. Fournet. Modern Concurrency Abstractions for C#. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, pages 415–440, June 2002.

[6] D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36:90–102, September 1993.

[7] D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10(11–13):1043–1061, September 1998.

[8] D. Caromel and J. Vayssière. Reflections on MOPs, Components, and Java Security. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, pages 256–274, June 2001.

[9] D. Chatterton. *Dynamic Dispatch in Existing Strongly-Typed Languages.* PhD thesis, School of Computer Science & Software Engineering, Monash University, Australia, June 1998.

[10] S. Chiba. Loadtime Structural Reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, pages 313–336, June 2000.

[11] A. Eliasson. *Implement Design by Contract for Java using Dynamic Proxies.* http://www.javaworld.com/javaworld/jw-02-2002/jw-0215-dbcproxy.html, February 2002.

[12] P.Th. Eugster and S. Baehni. Abstracting Remote Object Interaction in a Peer-to-Peer Environment. In *2002 Joint ACM Java Grande - ISCOPE Conference*, pages 46–55, November 2002.

[13] P.Th. Eugster, R. Guerraoui, and C.H. Damm. On Objects and Events. In *Proceedings of the 16th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001)*, pages 131–146, October 2001.

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

[15] M. Golm and J. Kleinöder. Jumping to the Meta Level: Behavioral Reflection Can Be Fast and Flexible. In *Proceedings of the 2nd ACM International Conference on Metalevel Architectures and Reflection (Reflection '99)*, 1999.

[16] L. Gong. *Inside Java 2 Platform Security: Architecture, API, Design and Implementation*. Addison-Wesley, 1999.

[17] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[18] U. Hölzle. Integrating Independently-Developed Components in Object-Oriented Languages. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP '93)*, pages 36–56, July 1993.

[19] JBoss. *JBoss 3.0*. http://www.jboss.org, 2003.

[20] Kalixia. *jAdvise*. http://www.kalixia.com/weblogs/space/jAdvise, 2003.

[21] R. Keller and U. Hölzle. Binary Component Adaptation. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP '98)*, pages 307–329, July 1998.

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

[23] G. Kiczales, J. Lamping, A. Menhdhekar, Ch. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, pages 220–242, June 1997.

[24] S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98)*, pages 36–44, October 1998.

[25] H. Liebermann. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86)*, pages 214–223, September 1986.

[26] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2 edition, 1999.

[27] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the 2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pages 147–155, October 1987.

[28] S. Matsuoka and S. Kawai. Using Tuple Space Communication in Distributed Object-Oriented Languages. In *Proceedings of the 3rd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '88)*, pages 276–284, November 1988.

[29] B. Meyer. Applying Design by Contract. *IEEE Computer*, 25(10):40–51, October 1992.

[30] M. Odersky, E. Runne, and Ph. Wadler. Two Ways to Bake Your Pizza - Translating Parameterised Types into Java. In *Generic Programming '98*, pages 114–132, 2000.

[31] D.C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. 2000.

[32] Sun. *Adding Generics to the Java Programming Language. Java Specification Request (JSR) 000014*.

[33] Sun. *Dynamic Proxy Classes*, 1999.

[34] Sun. *Java Core Reflection API and Specification*, 1999.

[35] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A Bytecode Translator for Distributed Execution of "Legacy" Java Software. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, pages 236–244, June 2001.

[36] H. Lam Th. Thai. *.NET Framework Essentials*. O'Reilly and Associates, Inc., June 2001.

[37] I. Welch and R. Stroud. From Dalang to Kava: the Evolution of a Reflective Java Extension. In *Proceedings of the 2nd ACM International Conference on Metalevel Architectures and Reflection (Reflection '99)*, pages 2–21, 1999.

[38] I. Welch and R.J. Stroud. Kava-Using Byte Code Rewriting to Add Behavioural Reflection to Java. In *Proceedings of the 6th Usenix Conference on Object-Oriented Technologies and Systems (COOTS'01)*, pages 119–130, January 2001.