Uniform Proxies for Java

Patrick Eugster
Department of Computer Science
Purdue University
p@cs.purdue.edu

Abstract
The proxy abstraction has a longlasting tradition in object settings. From design pattern to inherent language support, from remote method invocations to simple forms of behavioral reflection – incarnations as well as applications of proxies are innumerable.

Since version 1.3, Java supports the concept of dynamic proxy. Such an object conforms to a set of types specified by the program and can be used wherever an expression of any of these types is expected, yet reifies invocations performed on it. Dynamic proxies have been applied to implement paradigms as diverse as behavioral reflection, structural conformance, or multi-methods. Alas, these proxies are only available “for interfaces”. The case of creating dynamic proxies for a set of types including a class type has not been considered, meaning that it is currently not possible to create a dynamic proxy mimicking an instance of a given class. This weakness strongly limits any application of dynamic proxies.

In this paper we unfold the current support for dynamic proxies in Java, assessing it in the light of a set of generic criteria for proxy implementations. We present an approach to supporting dynamic proxies “for classes” in Java, consisting in transformations performed on classes at load-time, including a generic scheme for enforcing encapsulation upon field accesses. These transformations seemlessly extend the scope of the current support for dynamic proxies. We discuss the precise benefits and costs of our extension in terms of the criteria introduced, and illustrate the usefulness of uniformly available proxies by implementing future method invocations both safely and transparently.

Categories and Subject Descriptors D2.3 [Software Engineering]: Coding Tools and Techniques—Object-oriented programming; D3.3 [Programming Languages]: Language Constructs—Classes and objects

General Terms Experimentation, Languages

Keywords Java, future, proxy, transformation

1. Introduction
The concept of proxy — an object mimicking another object — has a longlasting tradition in the realm objects, enjoying both innumerable incarnations and applications. 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”) [9], for instance, are probably among the most prominent of all design patterns. Examples of widespread and traditional applications of proxies are of course remote invocations [24], future objects in asynchronous, so-called future, invocations [34] (remote or not), and behavioral reflection [16].

At version 1.3, Java’s core reflection API [26] has seen the addition of dynamic proxies. A dynamic proxy is a typed proxy, created at run-time for a type (a set of types) defined by the application. Such an object can be used in a consistent manner wherever an expression of that type (of any 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 applications. Examples are plentiful. The WWW hosts various reports on implementations of popular paradigms in Java based on dynamic proxies, e.g., implicit (structural) conformance, future invocations, dynamic multi-dispatch (a.k.a. multi-methods), design by contract [21] or aspect-oriented programming [17] (see for instance [7] and [13] respectively for the latter two). Furthermore, dynamic proxies are now officially endorsed as the preferred means for implementing Java RMI, making the rmic pre-compiler obsolete.

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. This requires no specific support from the Java compiler or virtual machine [25]. Leaving aside the proxy-inherent two-body issue (a proxy and the object it mimics remain distinct entities) this simple solution unfortunately manifests important limitations: dynamic proxies are not uniformly available, but only “for interfaces”. In other terms, such proxies can not be assigned to variables whose static type is a class (type). This limitation strongly hampers the potential of dynamic proxies overall. To fully exploit the above-mentioned paradigms through implementations based on dynamic proxies, programs are ultimately constrained to define all variables as being of interface types, and to use classes only for instantiation. Consider the case of future invocations, which intuitively make an ideal case for dynamic proxies. Unless respecting the above-mentioned severe constraint, they can currently only be implemented explicitly [34] without such proxies. Transparency can be achieved of course by other means. [22] for instance makes use of a powerful static analysis and corresponding program transformations, which however only deal with future invocations, and are not complete.

The goal of this paper is to make the concept of dynamic proxies in Java more uniformly available, i.e., also “for classes”. Unlike
We unfold the current support for dynamic proxies and their implementation in Java.

We discuss limitations of proxies overall, and in the case of Java, and establish a set of criteria to express what we expect from a “proxification” scheme. This set includes concerns such as completeness, safety, security, transparency, or performance overhead.

We propose uniform dynamic proxies for Java, building on the existing support for dynamic proxies. This leads to creating a dynamic proxy class for a set of types including a class \( C \) as subclass of \( C \). In order to make such an extension approach feasible, we introduce a set of byte code transformations including a general scheme for transforming instance field accesses to invocations of automatically created getter/setter methods.

These transformations yield an internal uniformly virtual object model, in which any type/member can be extended/overridden. We present how we have tamed the power of this model in the loading process to ensure extension and visibility constraints of programs.

We illustrate the benefits of uniform dynamic proxies by implementing future method invocations in a way balancing the transparency of [22] and the safety of [31], without requiring any future-specific static analysis and program transformations.

We discuss our uniform proxies/uniformly virtual object model in the light of the proxification criteria introduced. This includes measurements (conducted with the SpecJVM benchmark suite) of the overhead introduced by our uniformly virtual object model, which leads to quantifying the cost of encapsulation. We investigate the pros and cons of (different faces of) transparency in more depth through the case of future invocations, discussing also ways of dealing with the inherent two-body issue.

Roadmap. Section 2 dissects the original concept and implementation of dynamic proxies in Java. Section 3 discusses the limitations of dynamic proxies, and introduces our criteria for proxification. Section 4 presents our approach to supporting dynamic proxies for classes, based on program transformations performed at byte code level. Section 5 presents various issues tied to the implementation of these transformations in the class loading procedure. Section 6 reflects on safety and transparency in future calls, and illustrates how uniform proxies can help provide both. Section 7 discusses various issues, such as security impacts and limitations, further in the light of our proxification criteria. Section 8 overviews related work. Section 9 concludes with final remarks.

2. Unfolding Java Proxies

This section dissects the concept of dynamic proxies introduced with Java 1.3.

2.1 Presentation

This appraisal includes a presentation of the types involved (see Figure 1) and perceived by programers when manipulating dynamic proxies, and the creation of dynamic proxy classes. For presentation simplicity, we henceforth drop the qualifier “dynamic” when referring to proxies in the sense of Java reflection, unless confusion might otherwise arise. Furthermore, we omit package names when they are unambiguous (e.g., java.lang.reflect common to most types for reflection except class meta-objects).

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

public class Proxy implements Serializable {
    protected InvocationHandler h;
    protected Proxy(InvocationHandler h) { this.h = h; }
    public static InvocationHandler getInvocationHandler(Object proxy)
        throws InvocationTargetException, IllegalAccessException {
            InvocationHandler h;
            public static Class getProxyClass(ClassLoader loader, Class[] interfaces)
                throws InvocationTargetException, IllegalAccessException {
                    Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h)
                        throws InvocationTargetException, IllegalAccessException {
                        ...
        
Figure 1. Types InvocationHandler and Proxy

and abbreviate syntax and names. For example in stands for the keyword implements, and RuntimeException might be used instead of java.lang.RuntimeException. Classes and interfaces presented in separate figures are reduced to those parts which are relevant for the present work.

2.2 Proxy Objects

A proxy is an object which conforms to a non-empty set of interfaces \( \{I_1,...,I_n\} \), for which that proxy’s class was created. The corresponding proxy class extends class Proxy [27] depicted in Figure 1, and implements all interfaces \( \{I_1,...,I_n\} \) (see Section 2.4). Conforming to all those types, a proxy can be cast to any of them, and hence any method defined in either of those interface can be invoked on the proxy.

2.3 Invocation Handlers

Every proxy object has an associated object of type InvocationHandler (see Figure 1), which handles the method invocations performed on the proxy. More precisely, these invocations are reified and passed to the invocation handler through its invoke method as illustrated by Figure 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 reification of the method (an instance of method Method) that was invoked on the proxy, and (3) the actual arguments (as an array of Objects). 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, and it is declared to throw instances of Throwable.

According to the specification [25], an exception of type ClassCastException is thrown if a wrong type is returned by invoke. Any exception from invoke of a type not declared by the actually invoked method is wrapped in a UnknownAccessException. This occurs also upon invocation of a method \( m \) on a proxy created for a set of interfaces \( \{I_1,...,I_n\} \) in which at least two interfaces \( I_i \) and \( I_j \) declare \( m \), but with non-identical sets of exceptions. To ensure conformance, \( m \) will namely be declared to throw only exceptions declared by \( m \) in both \( I_i \) and \( I_j \). A NullPointerException is thrown if null is returned by the invoke method but the return type of the method invoked on the proxy is a primitive type (values of such types being wrapped by corresponding object types).
getProxyClass
what to perform upon method invocation
implemented by meta-objects representing methods. While the latter
method is a class loader, with which the possibly created class is to
implements those interfaces. A further argument to the above
class has already been created for that precise set) as a class which
When invoked, the method creates a proxy class, directly as byte
argument is omitted for simplicity):

\[ \mathcal{P}[I_1, \ldots, I_n, \text{ih}] \]

The general notation \( \langle \cdots \rangle \) represents the reification of con-
structs (see Figure 3), and the notation \( \mathcal{P}[\cdots] \) represents the
“proxification” of constructs.

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 [25]. Essentially, the procedure fails 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 consid-
ered part of method signatures in Java), or for two non-public in-
terfaces 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 the package in the latter
case to be always the same, and simply denote it as \( p \).

2.4 Proxy Creation

The `getProxyClass` method in class `Proxy` expects as argument
a set of interfaces defined by their respective `class` meta-objects.
When invoked, the method creates a proxy class, directly as byte
code, as a class implementing that set of interfaces (unless 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 the `getProxyClass` method described above, the
`newProxyInstance` in addition instantiates the possibly
generated proxy class. It hence takes an additional argument,
which is an invocation handler, and returns an instance of that
class which the specified invocation handler is associated with.
Supposing a set of interfaces \( \{I_1, \ldots, I_n\} \) and an invocation handler
\( \text{ih} \) we abbreviate such a call in the following (the class loader
argument is omitted for simplicity):

\[ \mathcal{P}[I_1, \ldots, I_n, \text{ih}] \]


ewProxyInstance(...) \[ \{I_1, \ldots, I_n\}, \text{ih} \]

The general notation \( \langle \cdots \rangle \) represents the reification of con-
structs (see Figure 3), and the notation \( \mathcal{P}[\cdots] \) represents the
“proxification” of constructs.

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 [25]. Essentially, the procedure fails whenever
conflicts would also arise if a class was explicitly, i.e., statically,

The static notation `T.class` is preferred to the dynamic lookup
`Class.forName("T")` for brevity, even when `T` is not known statically,
and thus the former one would not apply. Similarly, the quotation marks `"` will
be omitted in certain cases.

\[ \text{program } P := \{ \langle B \rangle \} \]
\[ \text{classes } B := c_1, \ldots, c_n \]
\[ \text{interfaces } D := I_1, \ldots, I_m \]
\[ \text{class } c := \{ F^B \}_{c_1} C \{ S \} \{ \langle I_1 \rangle, \ldots, \langle I_n \rangle \} \]
\[ \text{supertypes } S := \{ \text{ext } C \} \{ I_1, \ldots, I_m \} \{ \langle M^D \rangle \} \]
\[ \text{interface } i := \{ F^D \}_{I_1, \ldots, I_n} \{ \langle M^P \rangle \} \]
\[ \text{constructors } K := k_1, \ldots, k_n \]
\[ \text{variables } F^P := f_1, \ldots, f_n \]
\[ \text{methods } M^P := m_1, \ldots, m_n \]
\[ \text{constructor } k_i := \{ V^B \} \{ C \{ F^D \} \} \{ \text{the } X \{ e \} \}
\[ \text{variable } f_i := \{ V^B \} \{ T \}
\[ \text{method } m_i := \{ Q \} \{ V^T \} \{ T \} \{ \text{the } X \{ e \} \}
\[ \text{exceptions } X := C_1, \ldots, C_n \]
\[ \text{type } T := C \{ I \}
\[ \text{modifier } Q := \text{fin } (\text{virt}) \mid \text{abs}
\[ \text{visibility } V^H := \text{pub } (\text{pack}) \mid \text{prot } \mid \text{priv}
\[ \text{visibility } V^D := \text{pub } (\text{pack})
\[ \text{expression } e := \text{null } \mid \text{this } a \mid e.a \mid e.a = e
\[ \text{e.m(c, e.a, e.b), } P[I_1, \ldots, I_n, \text{ih}] \]

Figure 4. Simplified Java syntax

2.5 Proxy Class Internals

Once these cases have been ruled out, the generation of a proxy
class for a given non-empty set of interfaces \( \{I_1, \ldots, I_n\} \) occurs as
described in more detail in the following.

We consider a simplified Java syntax, outlined in Figure 4. Figure
5 summarizes relationships and orders (partial) among types
(\(<\) ), and (total) for visibility (\(<\) ), and modification (\(\leq\)) qual-
ifiers. We also introduce an order among (signatures o) methods
(\(\leq\)), as well as a union operator (\(\cup\)) and a union set \(\{\} \)
capture overriding. \( (x)_{T} \) denotes the declaration of \( x \) in \( T \). Based
on these definitions, \([I_{\text{MSET}}]\) and \([I_{\text{MSET-REC}}]\) in Figure 5 de-
scribe the construction of the sets of relevant methods \( M^D(T) \) for
a given type \( T \), also under consideration of supertypes (\( M^P(T) \)).
Only interfaces will be considered for now; the case of classes and
hence fields will be introduced in Section 4.

The rules outlined in Figure 6 describe the original creation of
proxies in source code notation for readability. When a proxy class
is created for interfaces \( \{I_1, \ldots, I_n\} \), a class is created (here called
\( I_1, \ldots, I_n, \text{Proxy} \)), which contains two constructors, and a method
body for each method in \( \bigcup_{c \in C} M^P_{\text{set}}(T) \). The body of such a proxy
method is generated according to Figure 7.

Note that methods `equals`, `hashCode`, and `toString` inherited by
every class from `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 `Object` are not overridden by
proxy classes, as they are `final`.

1 The static notation `T.class` is preferred to the dynamic lookup
`Class.forName("T")` for brevity, even when `T` is not known statically,
Consider the interface \( i \), defining a single method:

```java
import java.io.*;
public interface I {
    public String foo(Integer i) throws IOException;
}
```

The following lines illustrate the creation of a proxy for this interface \( i \). The invocation handler associated with the proxy simply prints the name of methods invoked on the proxy to the standard output, and forwards the invocation to another object \( realI \) implementing \( i \):

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

Figure 8 outlines code of the proxy class generated for \( i \) in a schematic manner, chosen to accomodate our own implementation dealing also with the issues pointed out in the next section(s).

Note that the class name is arbitrarily chosen: according to [25], the name space "Proxy" is reserved. Code fragments specific to custom interface \( i \) and its method(s) are emphasized. Methods from Object which are overridden are omitted in Figure 8 for brevity.

### 3. Assessment

In this section, we elaborate on the limitations of proxies in general, and on dynamic proxies in Java in particular.
3.1 Proxy-inherent Limitations

The simplicity of the concept of proxy as an object able to mimic another object accounts for its wide adoption as well as for its relative ease of implementation in object systems – in the simplest form as design pattern. The same simplicity however accounts for its main limitation, the two-body problem: the proxy and the object it represents are distinct entities.

While in certain scenarios this distinction turns out to be useful, and even desired, it is restrictive in the case of behavioral reflection, and is one of the reasons why more recent derivatives of such reflection (e.g., [17]) avoid proxies. For associating a behavior through a dynamic proxy as a meta-level object with an existing base-level object, the invocation handler associated with the corresponding proxy would most likely have to be given a reference to that object, and henceforth, the proxy would always have to be used instead of the original object, for instance also for self-invocations. However, when a method of the actual target object invokes a further method of that very object, there is no way of intercepting that invocation ("self problem" [19]). Similarly, self-references returned by a base-level object invoked through a proxy would have to be recognized as such (e.g., by the associated invocation handler in the case of Java’s proxies), such that again a proxy could be returned instead ("encapsulation problem" [19]).

The detecting and handling of all such situations is very hard. Base-level objects can for instance also return references to their fields which are difficult to identify, but would have to be shielded behind proxies as well.

3.2 Java Proxies

In Java, 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, etc. When striving for a uniform model of behavioral reflection with the current support for proxies in Java, the programmer basically would be constrained to program exclusively with variables of interface types, and make use of classes only in instantiations.

This limitation applies more generally to any application which would like to proxify objects of arbitrary types.

3.3 Extension Approach

Achieving uniform proxies (i.e., any object can be proxified) is of course possible with inherent language support, or in "straightforward" type systems. A proxy can be created for any object in Smalltalk as an instance of a generic proxy class overriding just the forward object methods. A proxy can be created for any object in Smalltalk as an instance of a generic proxy class overriding just the forward object methods. A proxy can be created for any object in Smalltalk as an instance of a generic proxy class overriding just the forward object methods. A proxy can be created for any object in Smalltalk as an instance of a generic proxy class overriding just the forward object methods.

Members which are hidden might not be overridable. In Java, fields are shadowed [20], and thus do not undergo dynamic dispatch. Though the semantics and rationale are different, private methods are also in some sense hidden, and thus cannot be overridden by subclasses.

The coupling of initialization code between classes and superclasses, especially if not explicit, is another source of problems. In Java every constructor (except the no-argument constructor in the root class Object) must call a superclass constructor, whether this is explicitly coded as first instruction inside a given constructor, or a call to the default no-argument constructor of the superclass is automatically added [20]. Relying on such default constructors of subclasses for initialization of proxy classes can yield side-effects, and as a matter of fact, might also fail if such a constructor is hidden (i.e., private, see above).

3.4 Proxification Criteria

Based on the previous observations, we define in the following criteria of "proxification" considered for the extensions proposed in the following and their subsequent evaluation.

Completeness: This twofold criterion quantifies the proportion of the types which can be proxified at all, and also how much of those types can be proxified. Typically, the impossibility of creating proxies for final classes in Java, limits completeness in the first sense, while the impossibility of overriding certain members can limit completeness in the second sense.

Safety: Proxifying a construct, and giving the possibility to perform any actions instead of the ones defined by the original construct obviously can lead to safety issues ranging from unexpected exceptions to undesired side-effects.

Security: Closely related to the issue of safety is that of security: using an entity instead of another one can obviously also lead to security issues in a platform which incorporates security constraints.

Overhead: Any proxification, or direction used to achieve it, may introduce a performance overhead which must be considered.

Transparency: This measure describes how aware the programmer is of the points at which proxification occurs. The two-body issue nicely demonstrates the impact of transparency of a proxification mechanism.

These different properties are likely to be correlated in any proxy implementation, and often tradeoffs have to be made. The first criterion for instance describes the breadth of a proxification. Extending it beyond a certain point — even if possible — might however lead to unsafe or unsound behavior and thus be undesired. Similarly, transparency of a proxification mechanism is not actually desired in certain contexts as it may compete with safety. We will have a closer look at transparency later on in the context of future invocations.

4. Uniform Dynamic Proxies

This section proposes an enhancement of Java’s current implementation of dynamic proxies, aiming at increasing completeness by adding support for proxies for classes.

4.1 Uniformly Virtual Object Model

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. In order for a proxy to be able to reify any action performed on it, its class must hence “override” all superclass members (Figure 9 overviews the sets of members of classes). To deal with the limitations outlined in the previous section, we propose in the following a set of nearly independent classes. To deal with the limitations outlined in the previous section, we propose in the following a set of nearly independent classes. To deal with the limitations outlined in the previous section, we propose in the following a set of nearly independent classes. To deal with the limitations outlined in the previous section, we propose in the following a set of nearly independent classes. To deal with the limitations outlined in the previous section, we propose in the following a set of nearly independent classes.

4.2 Proxy Types

As a direct consequence, a proxy class created for a set of types including a class $C$ must subclass $C$ and hence cannot subclass the $C$ class. 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 10).

The Proxy class still serves as superclass for proxy classes created for a set of interfaces exclusively (see Figure 11), and hence implements ProxyType. This is depicted in Figure 12, which focuses on additions in the new backwards-compatible version of the Proxy class. Modifications from the original version are emphasized.

4.3 Access Handlers

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 methods $\text{get}$ and $\text{set}$ of an instance of that AccessHandler type associated with the proxy. The AccessHandler type hence complements the InvocationHandler interface. Such an instance must be passed to any proxy created for a class.

We thus augment our considered subset of Java of Figure 4 by a further kind of expression, leading to the following extended expression set:

\[
e_x = e \mid P[C, I_1...I_n, ah, ih]
\]

This new expression is a shortcut for creating proxies “for classes”. A proxy for a set of types including a class $C$ and set of interfaces $\{I_1...I_n\}$, an invocation handler $ih$, and an access handler $ah$, is created as follows:

\[
P[C, I_1...I_n, ah, ih] \triangleq \text{Proxy.newProxyInstance(..., \{C\}, \{\{I_1\},...,\{I_n\}\}, ah, ih)
\]

Consequently, class $Proxy$ has been added variants of the $\text{getProxyClass}$ and $\text{newProxyInstance}$ 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 $\text{Proxy}$ 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).

4.4 Field Accesses

Field accesses have to trigger method invocations in order to be reified. 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.

```
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 val)
  throws RuntimeException;
}
public class UnexpectedRuntimeException extends
  RuntimeException {...}
```

![Figure 10. New auxiliary types](image)

![Figure 11. Proxy types](image)

![Figure 12. Augmented proxy class](image)
Since in Java, like in many other languages, fields can not be overridden by subclasses, but only hidden (see Section 3.3), field access intercation can not be achieved simply by defining a getter/setter method pair à la `get/set` in a class `C` for each field `f` declared by `C`. Care must be taken that the information about the class `C` in which a field is declared is not lost. A solution to this consists in conveying information about the declaring class `C` of a field in its respective getter/setter method. To that end, we determine the name of such methods according to two respective functions \( G, S : \mathbb{A} \times \mathbb{B} \rightarrow \mathbb{M} \), where \( \mathbb{A}, \mathbb{B}, \) and \( \mathbb{M} \) denote the sets of field, class, and method names respectively. This is illustrated in the following through three recursive subclasses with \( G(a, C) = \text{get}\$C$a \) and \( S(a, C) = \text{set}\$C$a):

**Original classes:**

\[
\begin{align*}
\text{class } C1 & \{ \\
& \text{String } a; \\
& \} \\
\text{class } C2 & \text{ extends } C1 \{ \\
& \text{String } a; \\
& \} \\
\text{class } C3 & \text{ extends } C2 \{ \\
& \text{String } a; \\
& \}
\end{align*}
\]

**Transformed classes:**

\[
\begin{align*}
\text{class } C1 & \{ \\
& \text{String } a; \\
& \text{get\$C1\$s() \{ return } a; \} \\
& \text{void set\$C1\$s(String } s) \{ \text{this. } a = s; \} \\
\text{class } C2 & \text{ extends } C1 \{ \\
& \text{String } a; \\
& \text{get\$C2\$s() \{ return } a; \} \\
& \text{void set\$C2\$s(String } s) \{ \text{this. } a = s; \} \\
\text{class } C3 & \text{ extends } C2 \{ \\
& \text{String } a; \\
& \text{get\$C3\$s() \{ return } a; \} \\
& \text{void set\$C3\$s(String } s) \{ \text{this. } a = s; \}
\end{align*}
\]

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

**Original code:**

\[
\begin{align*}
& C1 \ c1 = \ldots; \\
& \ldots = c1.a; \\
& C2 \ c2 = \ldots; \\
& \ldots = c2.a; \\
& C3 \ c3 = \ldots; \\
& \ldots = c3.a
\end{align*}
\]

**Transformed code:**

\[
\begin{align*}
& C1 \ c1 = \ldots; \\
& \ldots = c1.get\$C1\$s(); \\
& C2 \ c2 = \ldots; \\
& \ldots = c2.get\$C2\$s(); \\
& C3 \ c3 = \ldots; \\
& \ldots = c3.get\$C3\$s();
\end{align*}
\]

This scheme ensures that always the right variable is accessed. The specific corresponding transformations in Figure 13, i.e., (10), (15), and (16), are dubbed \( T_{PA} \). Figure 14 shows the generation of field access methods for a given field (the generation of constructs is denoted by enclosing the source definition in \( \{[ \ldots ]\} \)).

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

Furthermore that for simplicity, the package name of a class is supposed to be part of the class name. 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), 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 \( p_1 \) and a class called another class of same name \( C \) in package \( p_2 \) of fields of the class are declared in another class of the same name \( C \) in package \( p_1 \) and both classes declare a field of same name.

![Figure 14. Generating access and stub methods, and constructors](image)

### 4.5 Private Fields and Methods

The above transformations do not enable the reification of accesses to fields which are private. This stems from the fact that the dispatch of a private (getter/setter) 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 [20]. To circumvent this caveat, getter/setter methods for private fields are defined with package visibility, the weakest visibility enabling overriding/dynamic dispatch.

The similarity between the lookup for private methods and the lookup for fields suggests the adoption of a scheme for interception of application-defined private methods inspired by the one applied for field accesses, consisting in complementing private methods with `stub methods`, through which former methods are invoked. A stub method differs from the original method in its visibility qualifier (package visibility) and name. Akin to getter/setter methods, stub methods convey information about their class and the method name. More precisely, stub method names are determined by a function \( U : \mathbb{M} \times \mathbb{B} \rightarrow \mathbb{O} \) (e.g., \( U(m, C) = C_{2m} \)). This avoids 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 methods declared to be `native` would invalidate lookup tables of corresponding native libraries.

The transformations \( T_{PM} \) corresponding to private methods in Figure 13 are (8) and (17). The transformation for private fields, \( T_{PF} \), involves transformation (11). \( M^B(T)[m(e_1 \ldots e_n)] \rightarrow \langle d \rangle^T \) denotes the method (declaration) \( d \) statically determined to handle a call to \( m \) with arguments \( e_1 \ldots e_n \) on a given type \( T \) (this does not involve an actual lookup in the transformations in Figure 13).

Note that \( T_{PF} \) only makes sense if \( T_{PA} \) is enabled. This is however the only dependency in the transformations presented in this section.

### 4.6 Final Classes and Methods

There is no magic behind the solution to circumventing the limitations introduced by the `final` keyword. 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 5.1). The schemes for dealing with final classes (\( T_{FC} \)) and final methods (\( T_{FM} \)) are described by (2) and (6) respectively. As a consequence of \( T_{FM} \) — 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 by proxies created for interfaces only.

---

[Note: The image at the bottom of the page contains a figure (Figure 14) illustrating generating access and stub methods, and constructors.]
The troubles with default constructors have been mentioned already.

4.7 Superclass Constructors

To the methods are still created in proxy classes according to instance methods of its superclass(es). Proxy methods for non-

- a public no-argument constructor. Every class is added a constructor with a single argument of that type, which simply passes that argument to the corresponding constructor of the superclass. 

4.8 Uniform Proxy Class Internals

Regarding the package in which a proxy class is created, the rules given in Section 2.4 applies without modifications. That is, a proxy class can only be created for a set of classes including a class of which some types have package visibility if they are indeed defined in the same package. Furthermore, proxy creation can fail in similar situations as 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.

4.9 Illustration

Suppose a class C implementing interface I introduced in Section 2.6 and adding a field bar:

- public String bar;
The following lines illustrate the creation of a proxy for C (the invocation handler ih from Section 2.6 is reused):

```java
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)Proxy.newProxyInstance(C.class.getClassLoader(),
  new Class[] {C.class}, null, ah, ih);
```

The proxy class created upon the invocation of the newProxyInstance method defined in the augmented Proxy class yields the pseudo code depicted in Figure 17 (with R() = ProxyInit).

5. Implementation Issues

This section discusses implementation choices and consequences of the proposed extensions.

5.1 Applying Transformations

As mentioned previously, the transformations described in the previous section are performed at load time. When a class C is to be loaded, it goes through the following components (see Figure 18): Analyzer: Class C’s byte code is analyzed, and the extension and visibility constraints of C conflicting with subclassing/overriding, e.g., occurrences of final or private, are identified. Verifier: The class C is verified, in particular against the constraints of previously loaded classes. Virtualizer: The transformations described in the previous section are applied. Beyond this point, C is uniformly virtual. Linker: The class is linked, and is ready to be used.

Though possible, this loading procedure has not been implemented as a user class loader, as the loading of only some classes in a way bypassing that loader (by using another user class loader, cf. Figure 18) would have strongly hampered safety and security [11]. Especially the possibility of enabling only certain transformations (e.g., only $T_{SC}$ and $T_{PA}$) would have introduced potential for misuse. Along the lines of several other “extensions” to Java (e.g., Agesen et al.’s proposal for genericity [1]), we make use of a pre-processor integrated with the virtual machine’s (default) class loading and linking path. More precisely, the analyser, verifier, and virtualizer are all parts of an instrumented system class loader. As illustrated by Figure 18, classes can still be loaded with user class loaders before being passed on to that system loader.

5.2 Increasing Safety

As mentioned previously, the parameterization of the functions $G(a, C)$ and $S(a, C)$ for naming field access methods, as well as of $U(m, C)$ for naming stub methods by the class C in which they are originally defined avoid accidental overriding in subclasses. The parameterization of constructors by $R()$ serves a similar purpose.

The functions described so far (e.g., $msC$ for a stub for m in class C) have however only been chosen for illustration. In fact, these functions also involve a secret key to hash m and C respectively. This measure not only further decreases the chances of accidental overriding of field access methods, stub methods, or constructors, but also avoids intentional overriding. Otherwise, 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 (see Section 3.4).
5.3 Resolving 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 8 and 17 namely, all relevant meta-objects (representing methods and fields respectively) are resolved (i.e., looked up) at every method invocation or field access targeting an instance of a proxy class. In practice, this repeated resolution is replaced by a static, lazy, one. More precisely, proxy class instances retain Method and Field objects once looked up in an internal table which is defined as static and thus shared by all instances of such a class.

Note at this point that the Method object looked up for a given method invoked on a proxy does not necessarily represent the exact method which would have been executed had the object not been a proxy. This is due to the fact that invocations of (non-private) methods are dynamically dispatched, and that the type of the variable through which the invocation is performed is “lost”. In particular, a proxy being called through a method \texttt{m} declared in two distinct interfaces \texttt{I}_1 and \texttt{I}_2 both implemented by the proxy class, can not distinguish whether it has been called through \texttt{I}_1 or \texttt{I}_2. Just like with the original implementation of dynamic proxies in Java, the method meta-object resolved upon that invocation is the one corresponding to the first of the interfaces \texttt{I}_1 and \texttt{I}_2 to have appeared in the array of types passed upon creation of the proxy (class). Classes take precedence over interfaces.

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 or 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).

5.4 Overhead

The transformations performed on code to support proxies for classes nicely illustrates the tradeoff between completeness and overhead. Removing the effect of the \texttt{final} keyword may affect performance by reducing the number of opportunities where the \texttt{just in time} (JIT) compiler can perform method inlining. However, JIT compilers such as Sun’s HotSpot™ 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 \texttt{TFC} and \texttt{TFM} in performance measurements. The main overhead associated with each of those transformations, and the only one in the case of \texttt{TSC}, becomes then the cost of performing the transformation itself, which is neglectable since classes are only instrumented once upon loading.

The transformations for field accesses (\texttt{TFA}) and private members (\texttt{TFM} and \texttt{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 observations are conveyed by Figure 19, 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 XE3, with a Pentium III processor, running Red Hat Linux release 7.3. (Though enabled, \texttt{TSC} is not mentioned in the figure, for the reason described above).

Note that interface invocations are still slow (this is not a necessity, but still 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 in an application, one would not only have to introduce corresponding interfaces for all types in an application as mentioned in Section 3.1, but would also experience the additional overhead of interface invocations.

6. Transparent and Safe Futures

This section illustrates the benefits of uniform proxies for Java by implementing future invocations transparently and safely. Various other application scenarios which have originally motivated this work, such as a lazy form of remote object passing, or the expression of predicates evaluated remotely in a deferred manner, are described in [8].

---

Figure 18. Class loading and delegation

Figure 19. Performance overhead of transformations

---
6.1 Explicit Futures in Java

When implementing futures [34] in Java, programmers are currently required to identify upfront which methods/subprograms might be called asynchronously, and are forced to make use of the type parameterized Future and Callable interfaces defined in package java.util.concurrent (see Figure 20, [27]). In short, type Future<T> is used on the caller side when the respective logical return type of an (asynchronous) invocation would have been T, and the called object will in fact implement Callable<T>. A programmer can achieve the asynchronous execution of a callable object by instantiating the predefined FutureTask with that object. Executing the future task (run) then leads to invoking the call method on the callable object. Any exception raised by the call is delivered to the caller at the point where it invokes the get method, wrapped up in an ExecutionException; otherwise the actual result is returned.

Figure 20. Interface Future and related types

6.2 Transparency

[22] proposes a powerful static analysis to make futures less explicit. More precisely, the authors strive for transparency, which we can split into the following three aspects:

TYPE: The (return) type of a future (call) appears to be the logical return type T of the method actually executed, and not Future<T> (of Callable<T> on the callee side).

IDENTITY: A future object, as placeholder for the value to be computed eventually, and the value then actually computed, appear as the same logical entity (cf. Section 3.1).

ASYNCRONY: The “lazy nature” of a future object is masked. It can be passed around like any other object, e.g., as argument to a method call, though the underlying asynchronous call computing that very object might not have completed yet. Any call on such a future object is then blocked until the computation followed up.

A specific library call Async.invoke(o.m(...)) is used to indicate future calls in programs (here on o) and guide the subsequent analysis which tracks potential occurrences of future objects, and wraps them in code to achieve the above transparency. Thus, a fourth form of transparency for future invocations is not a declared goal:

CALL: An invocation which is supposed to be performed asynchronously appears the same way as a synchronous invocation.

6.3 Safety

The authors of [31] observe that transparency of asynchrony, without supportive mechanisms, may hamper consistency. The concurrent execution of a future invocation and its continuation — the code between the future invocation and the actual access to the return value — can namely lead to a different observable behavior than the sequential execution of instructions the way they appear in the code. A flagrant example is given by an exception raised during a future call only after several actions have already been performed as part of the continuation.

The approach of [31] leverages on (optimistic) transactional mechanisms developed in [30]. A future call and its continuation are handled like two concurrent transactions, whose potentially conflicting actions (i.e., actions on shared data) must respect serializability [12]. If violations are observed, both the future call and its continuation are rolled back. [31] describes a class SafeFutures offering the same interface as FutureTask, but ensuring that the semantics of a forked future invocation executed in parallel with its continuation are the same as their sequential execution.

6.4 Safe Futures with Proxies

Based on this class SafeFutures and uniform proxies, we devise in the following an implementation of futures which combines the type transparency of [22] with the asynchrony transparency of [31], and adds call transparency, without requiring the specific program transformations of the former work. Figure 21 sketches the implementation. Class BackToTheFuture makes it possible to create a proxy for an arbitrary object, through which any method of that object can be (indirectly) invoked in an asynchronous manner transparently and safely. Assume for instance an instance of class C introduced in Section 4.9:

```java
C c = new C();
C cFut = BackToTheFuture.futurify<C>(c);
String s = cFut.foo(); /* future call */
System.out.println(s); /* synchronization point */
```

Calling futurify leads to creating a dynamic proxy with an instance of AsyncHandler for handling both method invocations and field accesses. While an action of latter kind is directly relayed to the proxified object, a method call triggers the instantiation of class SafeFuture with a SafeCall (a subtype of Callable), which will actually perform the call. A proxy representing the future object is returned, which is handled by an instance of FutureHandler. Any call on that proxy will then lead to blocking the caller on the SafeFuture until the underlying future call has completed.

The transparency achieved with this solution is discussed in more depth in Section 7.4.

7. Discussion

This section discusses various issues, such as security implications and limitations of the original and uniform proxies in the face of the criteria introduced in Section 3.4.

7.1 Security

No programming language has so far undergone as intensive investigations in terms of security as Java. Core notions in Java security are protection domains, permissions, and policies [11]. Protection domains correspond to certificates for signing classes, and/or URLs for obtaining classes, and

---

2 Rollback capabilities can be achieved in other less intrusive ways, cf. [18].
3 An in-depth presentation of security issues raised by general-purpose reflective extensions to Java can be found in [5].
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 by classes on the execution stack at that point include 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 the 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 an untrusted party with direct access to (instances of) a class, that party can be urged to access (instances of) that class through dynamic proxies. The corresponding handlers can then at run-time decide on granting permissions or not.

7.2 Introspection and Uniformity

Ensuring that introspection objects (instances of, e.g., Class, Method) representing the structures of linked classes do not reflect changes made at load-time can improve safety. A deeper integration of our uniform virtual object model in the Java virtual machine could go hand-in-hand with instrumented introspection classes. Such instrumentations would also apply to related approaches where corresponding issues are however rarely addressed.

7.3 Primitive Types

Dynamic proxies can not be created for primitive types. One might argue that this is not a limitation of dynamic proxies, but rather a consequence of Java’s hybrid type system. In addition, since primitive types do not have any members, there might be no need to intercept/override any calls to values of primitive types at all. The case of future invocations however nicely illustrates that the lack of proxies for primitive types does sensibly reduce completeness: methods to be invoked in an asynchronous manner can not have primitive return types.

A common workaround for this problem consists in introducing own wrapper classes for primitive types, which define methods corresponding to the operators that apply to them. 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 on and exploited, infringe license terms.

Note that the automatic boxing/unboxing of values of primitive types now in Java 1.5 only slightly alleviates the restricted completeness. Values of primitive types may for instance be boxed automatically by objects, but since this occurs transparently to the programmer there is no way of proxifying these objects in many situations.

7.4 Transparency in Future Invocations

We believe that the explicit “futurization” of an object with our futures presented in the previous section, alike in [22], is not a drawback. It reminds the programmer of the restrictions that still apply, such as with asynchronous exceptions. In fact, the solution of [31] still has some limitations with that respect. Take the case of a try...catch block around a future call, intended at handling exceptions potentially caused by that call. The end of such a clause should bound the call’s continuation – otherwise an exception might be thrown “too late”. In the worst case, the future object is a return value for the enclosing method. This case is not considered at all by many future implementations which strive for asynchrony transparency. An explicit proxification, in contrast, can also offer the possibility of registering an asynchronous exception handler for such a scenario.

The identity transparency which [22] strives for can be approached by adding a transformation, outlined below, which transforms the equals method for object comparisons such as to forward any such call to the object compared to, in case that object is a proxy.

$$T[pur boolean equals(T a) \{ e \}] =$$
$$public boolean equals(T a) \{$$
$$\quad if (a instanceof ProxyType) return a.equals(this);$$
$$\quad T[e]$$
$$\}$$

That way, a proxy handler such as AsyncHandler in Figure 21 which stores a reference to its associated proxified object can simply compare the argument with that object. This transformation works even in the presence of calls to superclasses within an equals method. Alas, a remaining drawback is the fact that the equals method is not necessarily symmetric anymore, as recommended [27]. Also, the == operator will still reveal whether two
variables point to the exact same object or not (while comparisons
based on the hashCode method in Object can be influenced by having
proxies/invocation handlers define the result). More intrusive
transformations could be introduced to deal with those cases, e.g.,
by wrapping such operator occurrences (cf. Section 8.1).

8. Related Work
The work closest to ours can be found in the area of behavioral
reflection. We discuss in the following these efforts4, and also a
closely related effort on the topic of future invocations for Java.

8.1 Kava
Kava [33] is a general extension to Java reflection providing behav-
ioral reflection, relying on a specific user class loader to modify
classes at load-time. Kava however follows an approach (unlike its proxy-based predecessor Dalang [32]), 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 transform the lines (source code for readability)

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

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

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

Such an approach enables the 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
and method invocation through a proxy. In contrast, with our
approach, such expensive calls to the core reflection API are made
at most once for a same method or field for all uses of that member
(see Section 5.1).

The use of an instrumented user class loader can be bypassed
(see Section 5.1), thus putting the uniform application of reflection
at stake [32]. Just like in our case, transformations 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).

8.2 Javassist
Javassist [6] is another extension to Java reflection, promoting load-
time structural reflection. Javassist offers a core API operating
at byte code level, and a more high-level API providing useful
“macros” built on former one (e.g., addition or modification of
methods), including a specific classloader for the instrumentation
of classes.

Javassist is extremely general and powerful, and has many po-
tential applications. Behavioral reflection is in fact only one 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

---

4 See [33] and [5] for more broader and detailed surveys of existing reflec-
tive extensions to Java.

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 replac-
ing field accesses by invocations, however to class methods.

The general applicability of Javassist has been illustrated by
realizing binary code adaptation [15], aspect-oriented program-
ming [14], or a form of synchronous RMI without static proxy
generation. Based on the latter experience, Addistant, another in-
stantiation of Javassist, is described in [28]. Addistant aims at the
distribution of “legacy” Java programs, that is, Java programs
developed without distribution in mind.

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 [28]) is also discussed. The problems
with final classes and methods, as well as constructors, are pointed
out, unlike the cases of private methods and field accesses.

Javassist has been used to implement a first prototype of the
transformations presented in this paper.

8.3 ProActive
ProActive, a descendant of JavaVL (“Java parallel”), is similar to Ad-
distant, in that it aims at providing features for “transparent” dis-
bursed or parallel execution of Java programs [4]. ProActive ad-
vocates the use of implicit futures, to decouple remotely interacting
components, through proxies obtained at run-time by manually in-
stantiating proxy classes part of the ProActive libraries. ProActive
could thus definitely directly benefit from uniform proxies.

9. Conclusions
This paper has presented an approach to broadening the scope of
Java’s own concept of dynamic proxies.

The solution presented in this paper neither makes use of a spe-
cific compiler nor relies on dispatch instrumentation in the virtual
machine, but can do with a set of transformations performed at class
loading.5 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 trans-
formation scheme whose applicability is not limited to the gener-
ation of dynamic proxies and the Java language.

We have discussed tradeoffs of proxification, such as between
completeness, safety, and security, and have proposed a simple way
of dealing with the inherent two-body limitation of proxies. Given
the generic applicability of the proxy abstraction, the set of poten-
tial applications which can benefit from our extension is unlimited.
We have illustrated our uniform proxies by implementing future
invocations both transparently and safely, further dividing trans-
parency into different aspects and pointing out which of these as-
pects are actually desirable in what contexts.

The main remaining drawback seems to be a consequence of
Java’s hybrid type system. In fact, it’s not surprising that the ab-
sence of a uniform object model makes it hard to achieve a uniform
proxy model. 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 our work could also be extended to Microsoft’s
.NET platform [29], which proposes a closely related concept of
dynamic proxies with nearly the same limitations as in Java. For

5 This makes a case against the claim that the achievement of field access
interception with a proxy approach is impossible without specifically mod-
dified virtual machine [5].

6 Not to be confused with the Kava approach to behavioral reflection in
Java [33], cf. Section 8.1.
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.

Acknowledgments

We thank Christian Damm, Sébastien Bachni, and the anonymous referees for helpful comments on draft versions of this paper, as well as all those who contributed to the implementation of this work.

References


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


