

## Hygienic methods — Introducing HygJava

**Jarosław D. M. Kuśmierek**, Institute of Informatics, Warsaw University

*Partly supported by the Polish government grant 3 T11C 002 27*

*and by SOFTLAB - Poland, Warsaw, Jana Olbrachta 94*

**Viviana Bono**, Dept. of Computer Science, Torino University

*Partly supported by MIUR Cofin '06 EOS DUE project*

One of the base concepts of object-oriented programming is that of “method”.

In languages supporting inheritance, this concept is realized by three different actions: (i) the introduction of a new method; (ii) the implementation/override of an existing method; (iii) the method call.

The bindings between (ii) and (i), and between (iii) and (i) are typically based on method names, which are not guaranteed to be unique, thus such bindings might cause some ambiguities. As a result, modifications of existing code may cause errors in some other parts of the code, especially in programs written by third party developers; overall, a programmer cannot predict the moment in the execution when such ambiguities will arise.

In this paper, we describe the nature of these problems and propose a general mechanism to overcome ambiguities in a safe, straightforward, and flexible way. To study the details of this mechanism, and make the reader more familiar with it, we show how to apply this mechanism to Java, and also to a mixin-oriented language called MixedJava.

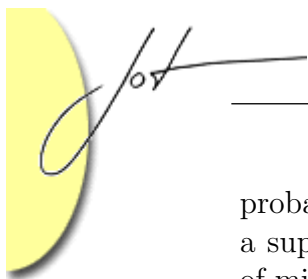
### 1 INTRODUCTION

The base object-oriented concept of “method” is realized by three different actions: (i) the introduction of a new method; (ii) the implementation/override of an existing method; (iii) the method call.

The bindings between (ii) and (i), and between (iii) and (i) are typically made by method name, which is not guaranteed to be unique, thus such bindings might cause some ambiguities. Additionally, in many popular languages (like Java and C#), the distinction between (i) and (ii) is also based upon names.

Therefore, modifications of existing classes (even modifications designed as conservative extensions of some functionality) may cause errors in some other part of the code referencing such classes, especially in programs written by third party developers. In general, a programmer cannot predict the moment in the execution when such ambiguities may occur.

Moreover, in languages containing a *mixin* construct, the set of allowed combinations of modules is much bigger, thus all these ambiguity problems are more



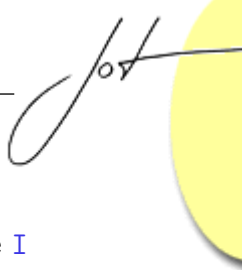
probable to occur. Recall that a mixin is a subclass parametrized with respect to a superclass, and mixin inheritance is obtained by applying a mixin (or a “chain” of mixins) to a class. Mixins were first introduced informally in a dynamically type checked language called Flavors, [14], but then developed in a statically typed language by Gilad Bracha, [6, 5]. A mixin can introduce new methods, request some methods to be supplied by its superclass, and override some of the superclass methods as well. Mixin application must obey some type constraints, in order to insure that the mixin and the superclass “agree”. Therefore, during mixin application, name clashes can occur.

This paper proposes a solution to overcome the problems introduced by modifications of code in the presence of the name ambiguities described above, and it is organized as follows. In Section 2 we present the kinds of problems which occur in both a class-based environment and in a mixin-based environment. In Section 3 we present some instances of problems occurring in the Java API’s due to name clashes, in order to give some evidence for the relevance of the ambiguity-related problems. In Section 4 we introduce our solution to those problems. In Section 5 we present the *HygJava* and *MixedHygJava* languages, respectively as modifications of the Java language (as a class-based representative), and of MixedJava (a mixin-based prototype language by Flatt et. al [8]). Those modifications do not suffer from any of the problems described in Section 2. In Section 6 we present the semantics of our approach by supplying a translation from HygJava to Java, and analyze the type system of such a language. In Section 7 we present a way of integrating such a new approach in an existing language like Java. This way, we can promote a development of safer code, while retaining compatibility with existing code. In Section 8: (i) we present some ideas about development tools for making the writing of safe code easier, according to our approach; (ii) we present the related works.

## 2 THE ORIGINS OF THE PROBLEMS

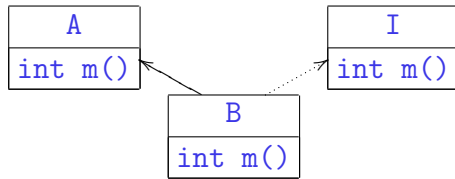
In this section we present three kinds of ambiguity problems which can occur when programming in a Java-like language, that are dealt with in this paper. The first two of these problems occur within the Java language (and also, with some differences, in other languages, see Section 8), while the last one occurs only in statically-typed languages containing a mixin construct.

**Name clash caused by the implementation of an interface.** Let us assume that class **A** and interface **I** are defined independently in different libraries (see the picture below). Assume also that both of these contain a declaration of method **m()**. Now, let us imagine that a developer needs to create class **B** as a subclass of class **A**, and also as an implementation of the interface **I**. Then it might happen that, in order to match the interface **I**, the implementation of method **m()** must be completely different from the one inherited from **A**. Therefore, the change of the implementation of method **m()** in order to have the behavior expected by **I** might



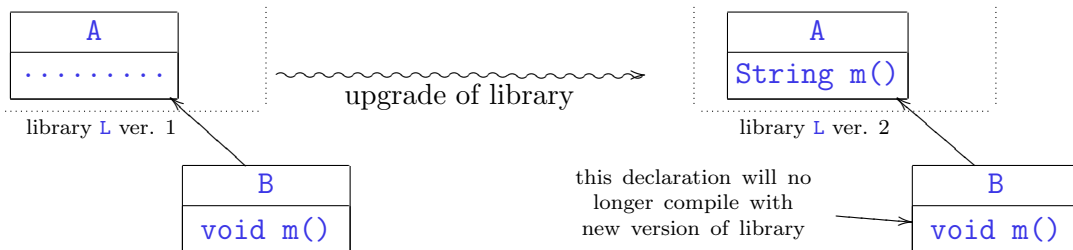
make the functionalities inherited from class **A** behave unexpectedly.

Moreover, if the result types of the methods **m()** introduced in **A** and interface **I** are incompatible, then the class **B** will not even compile.



**Name clash caused by the addition of a new method.** Let us assume that there exists a library **L** containing a class **A** (see the picture below). Assume also that there exists a class **B** created by a different developer as a subclass of class **A**, containing a declaration of a method **m()**. Additionally, let us assume that the developer of library **L** knows nothing about class **B**. Now let us assume that the developer of **L** decides to modify the functionality of **A** by adding a method **m()** (for example, by implementing a refactoring-based method extraction supported by a tool), and referencing it from existing methods. Then, unfortunately, class **B** used with the new version of **L** can suffer from two kinds of problems:

- If the result type of method **m()** in **B** is not compatible with the one declared in **A**, then **B** will not compile anymore.



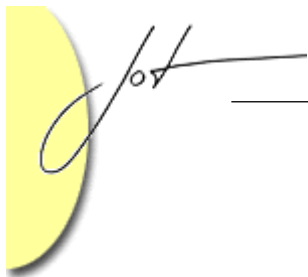
- If the result type is compatible, then **m()** in **B** will unexpectedly override **m()** from **A**, changing the behavior of the class **B** in a potentially undesired way. Consider as a more detailed example the one below, where class **A** is upgraded in the following way (and class **B** is declared as in the above figure):

```

class A
{ void oldmet()
  {  $\vec{I}_1$ ;
    if (...)
      {  $\vec{I}_2$ ; }
  }
}

class A
{ void oldmet()
  {  $\vec{I}_1$ ;
    if (...)
      m();
  }
  void m()
  {  $\vec{I}_2$ ; }
}
  
```

Here, the newly added method **m** is referenced from the existing method **oldmet**. Therefore, an accidental overriding of **m** will not only make class **B** not have the functionality expected from **m**, but also will change the behavior of another method (**oldmet**, in this case). Moreover, this dependency



between `m` and `oldmet` is not visible in the external interface of any class.

One common situation in which those problems may occur is the one when a core system is sold to many costumers and modified on the customer site, as well as upgraded during its lifetime.

This kind of problems are, in fact, the result of conflicting specifications of newly added and inherited method, and can be checked (dynamically or statically), if the specifications are formally defined as assertions and verified. This can be done, for example, in the Java environment with the use of the tool *JML*, [7], or in Eiffel via the *Design by Contract*, [12]. Both warn the programmer with a "specification-not-fulfilled error", Eiffel at run time, JML both at verification time and at run time, allowing the detection of conflicts that must be fixed in order to make the program work properly (and therefore eliminate the warnings).

**Name clash caused by mixin application.** Let us assume that there exist two independently developed mixins `M` and `N`, both adding method `m1` with the same name and types of parameters and result. Assume also that there exists class `A` to which both of these mixins can be applied. Next, if we will build the class `M(N(C))`, then, depending on the implementation of mixins in the language, we will have either:

1. a conflict raised by the compiler, or
2. a class where an implementation from `M` overrides the one from `N` (such an approach is the one of Jam, see [4]), or
3. a class with both methods available, but only with one of them at a certain moment, depending on the context, given by the type of an object expression. If the variable has a type containing `M`, then method from that mixin will be accessible, and analogously for `N`. Such approach can be found in the MixedJava language [8].

We think that the third solution is the best of the described ones, however it is still not completely satisfactory, because: (*i*) in contexts where the receiving object expression has both types `M` and `N`, the choice of the method is still ambiguous; (*ii*) in some situations a programmer might need access to both methods in one single place.

Note that some of described problems can also occur with regard to public and protected field declarations. However, fields are less often declared with a public visibility, and cause ambiguity problems less often. Additionally, problems with fields are generally easier to solve (because fields cannot be redefined), hence these can be solved with basically the same techniques exploited to solve ambiguity problems concerning methods. Therefore, for the sake of simplicity, we decided to concentrate in most of this paper on methods only, but in in Section 8 we will describe the differences and similarities of solutions for dealing with field ambiguities with respect to the method related ones.



### 3 PROBLEMS IN THE JAVA BASE LIBRARY

In order to give some evidence for the relevance of the ambiguity-caused problems, we present an analysis of some Java API's (ver. 1.5) code, showing name conflicts that might lead to the problems described in the previous section. The Java API's of course compile and work; thus, the ambiguity problems which possibly occurred during the development of the Java API's themselves were dealt with traditional techniques, such as the renaming of some methods and by discarding some changes, which would be unnecessary if Java were a hygienic language.

First, we show some numbers representing the occurrences of the introduction of the same method name in different classes and interfaces (notice that we did not count overridden definitions, or implementations of methods declared in the interfaces). Then we will present some simple but representative examples to underline the nature of the problems caused by the lack of hygiene.

method	occurrences in			method	occurrences in		
	interf.	classes	total		interf.	classes	total
getName()	59	148	207	setName(String)	15	7	22
getType()	36	71	107	getAttributes()	13	33	46
close()	30	38	68	remove(int)	6	19	25
getLength()	28	33	61	setValue(String)	9	2	11
getValue()	24	45	69	setType()	10	1	11
item(int)	19	12	31	getWidth()	13	17	30
getId()	20	32	52	clear()	8	61	69
reset()	14	95	109	isEmpty()	8	32	40

Below we present three examples of possible problems deriving from the presence of the ambiguities described above:

- The `Set` interface contains the method `isEmpty` with the obvious meaning. The `Hashtable` class (which is a partial function that assigns a value to a value) contains a method `isEmpty` which checks if the dictionary contains any assignment. Unfortunately, if one wants to implement a set as a characteristic function of a subset, thus implementing it as a subclass of the `Hashtable` class, then it cannot be done because the meaning of `isEmpty` from the point of view of `Set` must mean that the `Dictionary` contains false assignments to everything, which does not mean that the `Dictionary` itself is empty.
- The `Map` interface (which represents the concept of mapping from one set to another) contains a method `clear`, which empties the mapping. The typical graphical component `List` (available in the `java.awt` package) represents a list of items, and contains a method `clear`, which makes this list empty. Now, let us assume we want to represent some mapping from some small domain (implementing the interface `Map`) as a visual component that displays mappings of the form `X -> Y` for each value `X` in our domain (and displaying `X -> ?` if no value is assigned to `X` in this particular mapping). We might then choose to make our component a subclass of `List`. Then, the meaning of the method

`clear` which will behave accordingly to the “contract” of interface `Map` should not delete all the items on the visible list, but just replace them with `X -> ?` which, unfortunately, is completely incompatible with the notion of clearing the visual list.

- The `java.sql.Connection` interface, which represents the concept of connection to an SQL database, contains a method `close`. Similarly, the class `java.net.Socket`, representing network communication sockets, contains a `close` method. Now, let us assume that someone would like to create a subclass of class `Socket` representing a socket designed especially for communication with some specific SQL database. Then we might want to implement the interface `Connection`. Unfortunately, closing the actual logical connection with the database does not have to imply closing the physical connection with the database (as a application might want to keep a pool of open physical connections, which can be used at any time when the logical communication with the database is needed, to make the connection quicker).

An alternative to inheritance which can be used to solve such problems is to keep the class containing the desired implementation as an internal component, and in some cases it might be a better design decision. However, it is bad if the lack of hygiene itself limits the possible uses of the inheritance mechanism in a language.

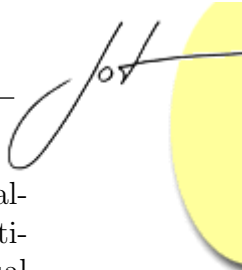
## 4 PROPOSED SOLUTION

### The core idea

In order to avoid the described problems, a language should be equipped with three *distinct* constructs: (i) method introduction, (ii) method implementation and (iii) method call. These notions must be defined as follows:

- *Method introduction*: the syntactic construct which declares the identifier of a new method. The method introduction consists of a method name, and a signature for the parameters, the result type, the list of possibly thrown exceptions, the visibility modifiers, etc. The language must equip implicitly the name of the method with the information about the place in which it was introduced. In other words, it must ensure that two method introductions placed in two different classes, or in two different interfaces, are always distinguishable.
- *Method implementation*: the declaration of the method body with a reference to a specific method introduction, to which this body must be bound. This implies that the syntax for method implementation must contain a precise and non-ambiguous reference to the corresponding method introduction. In the case of overloaded declarations, the reference is non-ambiguous when considered together with the signature of the parameters. Note that, in the case of Java, we do not have to deal with global methods (`static` methods), since those are statically resolved.





- *Method call*: an expression consisting of three parts: (i) the expression evaluating to an object on which the method is called, (ii) the method identifier referring to a concrete method introduction, and (iii) the list of actual parameters. Once again the reference to the method introduction must be non-ambiguous. In the case of overloading, the method call is non-ambiguous when considered together with the types of the parameters (as it is for the implementation).

In order to achieve the mentioned non-ambiguous references, we need unique identifiers for method introductions. The most straightforward solution to do this is to identify each method by its name together with its place of introduction, that is, its *context*.

We call such an approach *hygienic*. This name was inspired by the work of Allen et al. [3], which will be discussed in Section 8. We will talk about *hygienic methods* and *hygienic identifiers* to indicate non-ambiguous methods and identifiers, and we will call languages designed using such an approach *hygienic languages*. In particular the language resulting from applying this approach to Java will be called *HygJava*.

## HygJava

We present the idea of HygJava via the example shown in Figure 1. In this example, class **A** introduces a new method named `getName`, and supplies its implementation. This method is then re-implemented in class **C2**. However, what is interesting here is that class **C3** once again declares a new method of name `getName`, but this one is distinct from the one declared in **A**. Therefore, the implementation of the method in **C3** is non-ambiguously bound to this method declaration. Similarly, all method calls shown later can also be resolved non-ambiguously.

## MixedHygJava

Languages equipped with a mixin construct allow stronger reuse and composition of existing components, therefore the probability of ambiguities and clashes is much bigger. As an effect, the hygienic methods mechanism is even more important in such languages.

To exemplify, we borrow the syntax of MixedJava [8] to show our approach of combining hygienic method calls and mixins; we provide an example in Figure 2.

It is important to remark that the choice of MixedJava as our specimen for a mixin-based case was made, on the one hand, because MixedJava has a friendly syntax. On the other hand, MixedJava itself has an elegant approach for dealing with name clashes which we will compare to ours in Section 8, but the purpose of the present section is not adopting the MixedJava approach, just using an easy-to-use mixin-oriented syntax combined with our hygienic method approach.

```

package p;

class A
{ String getName();           //method introduction
  implement String p.A.getName() {...} //and its implementation
}
class B extends A
{ implement String p.A.getName()   //another implementation
  { ...; super (); ... }           //which overrides the previous one
}
class C extends A
{ String getName();             //second introduction of method getName,
  implement String p.C.getName() {...} //implementation of the local method
}

C obj = new C();
obj.p.A.getName(); //the method introduced in A and overridden in B is called
obj.p.C.getName(); //the method introduced in C is called

```

Figure 1: Example of HygJava code

```

package p.p2;

interface I
{ String met1();
}
interface J
{ Integer met1();
}

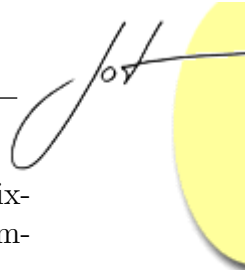
mixin M extends I
{ implement String p.p2.I.met1() {...}
}
mixin N extends I, J
{ implement Integer p.p2.J.met1() { ...; super(); ...}
}

class C implements I
{ implement String p.p2.I.met1()
  { ...; super();...}
}
class B extends C implements J
{ ... }
class D = M(N(C2)) //this class expression is ok

```

Figure 2: Example of MixedHygJava code





Similarly to the HygJava example, all methods implementations supplied in mixins refer to specific method introductions, so that combining mixins causes no ambiguity problems. Such a language will be called from now on *MixedHygJava*.

## 5 HYGJAVA SYNTAX

We will define now HygJava starting from the Java syntax. MixedHygJava syntax can be defined exactly the same way, that is, by applying the same modifications to the MixedJava syntax. To abstract from differences between those languages, we will use the term *modules* for classes, interfaces and mixins (the last one if we are considering MixedHygJava). As shown in the examples, we make the following modifications with respect to the original language, in order to obtain its hygienic version:

- We discard the Java method declaration (as it is in Java specification, [9]), introducing, instead, the two constructs defined below.
- We introduce a new member of a module, the *method introduction*, which has the same syntax as the *MethodHeader* nonterminal from the Java language specification, [9]:  
`[Modifiers] [TypeParameters] RetType Identifier ([FormalParameters]) Throws ;`
- We introduce another member of a module, the *method implementation*. It modifies slightly the syntax of a Java non-abstract class method declaration. The differences are: (i) the method implementation begins with the keyword `implement` to distinguish it from the method introduction; (ii) instead of the method name we use the non-ambiguous *HygienicIdentifier*:  
`implement [Modifiers] [TypeParameters] RetType  
HygienicIdentifier ([FormalParameters]) Throws MethodBody ;`  
According to the previously presented examples, the *HygienicIdentifier* is defined as a method name prefixed with a module name and a package name (of the package to which this module belongs to):  
`[PackageName.] ModuleName . Identifier`
- We modify the syntax of the method call. The method name is replaced with the appropriate hygienic identifier *HygienicIdentifier*.

## 6 SEMANTICS AND TYPE-CHECKING

### A semantics by translation

The semantics of HygJava can be expressed by a translation to Java. The same idea works for the interpretation of MixedHygJava into MixedJava.<sup>1</sup>

<sup>1</sup>The reader should notice that we do not translate it into Java, as there is no direct translation of MixedJava itself into Java.

Here there are the base concepts of this translation:

- We textually prefix the name of each method with its full path (a package and a module name), where dots are replaced with underlines. Such a method name will be called a *long-name*. This way we enforce the uniqueness of names and certify that no name clashes can occur. For example, the hygienic method identifier of the form `comp.pack.Class.met()` is replaced with the method name `comp_pack_Class_met()`. For the semantics, we assume that this translation is injective, in other words, we assume that in the HygJava sources there will be no method with name of the form `Class_met()`.
- Thanks to the fact that the long-name carries the information about the point of introduction, we can discard the introduction construct (unless it is an introduction without an implementation). The information whether it is a method introduction or the implementation of a method introduced elsewhere is included in the method name.

Hence, the actual translation is executed according to the following four steps:

1. Every introduction of a method placed inside a module in which there is also a implementation of this method is removed.
2. All remaining method introductions in classes (and mixins, if we are considering MixedHygJava) are replaced with `abstract` method declarations, where the names are replaced with the long-names. This is done according to the place of introduction, which means that the introduction of method `m` inside class `C` in package `p` is replaced with a declaration of method `p_C.m`. Introductions of methods in interfaces are transformed analogously, however without the `abstract` modifier.
3. Every HygJava implementation of a method is replaced with the ordinary Java method declaration. This is done by removing the keyword `implement` and replacing the hygienic identifier of the method with the long-name.
4. In every method call, a hygienic identifier (the one with a path) is replaced with a long-name.

The result of such a translation is Java code with non-ambiguous references. The Figure 3 shows the result of the translation of the HygJava example from Figure 1. Notice that a developer could program in such a style directly in Java. However, he will have no guarantee that a method name containing as prefix a class name will not be reintroduced anywhere else (by other developers not following this style), therefore the uniqueness cannot be enforced statically. Instead, in our hygienic approach, the uniqueness is implicitly guaranteed by the fact that each identifier is distinguished by its place of declaration.

## Performance of HygJava

HygJava has not been implemented yet. However, some performance measures for HygJava, implemented by manual translation into the Java language, are as follows:



```
package p;

class A
{ String p_A_getName() { ... }
}
class B extends A
{ String p_A_getName() { ...; super (); ... }
}
class C extends A
{ String p_C_getName() { ... }
}

C obj = new C();
obj.p_A_getName();
obj.p_C_getName();
```

Figure 3: Result of a translation from HygJava to Java

- The size of the Java code is enlarged by the prefixes containing the package name and the class name. A theoretical pessimistic upper bound is a quadratic growth in space of the Java files generated by the translation with respect to the size of the files written directly in Java (in a non hygienic way). However, the worst upper bound is reached in the cases where the name of package or the name of the class consume half of the source file. We performed some measurements, by rewriting a few medium-sized classes according to the translation previously defined. In practice, when names of packages and classes are of average length, our experiments show that the actual increase of length is not bigger than few dozens percent (for example, 15% in the case of the `java.util.Stack` class, and 10% in the case of the `java.util.HashSet` class).
- The size of the compiled `.class` files increases accordingly, except for the fact that `.class` files do not contain comments and keywords, thus they can be smaller. However, in practice, the hygienic `.class` files should be also no more than few dozens percent bigger than the original files.
- When it comes to the memory overhead, then it is insignificant, because the only difference is caused by the difference in the size of the loaded class files (which is described above). However, the cases when the memory usage really matters are the cases when a lot of memory is used by the heap, and from this point of view there is no difference between hygienic and non-hygienic code.
- The last issue is speed. However, except for the loading time of the classes when the actual linking of the identifiers is performed (which is rarely an issue), the speed of the running application is not modified at all, because all names are resolved during the linking.

## Another variant of the interface implementation semantics

One of the consequences of the semantics by translation defined in Section 6 is that when a class is declared to implement an interface it must either: (i) implement all the methods introduced in the interface (and its parent interfaces), or (ii) become an abstract class. A class does not implement implicitly the interface method introductions with the method implementations inherited from the parent class (as it is, instead, in Java).

Such a semantics is simple and clean, however some programmers may argue that it does not work the way they are used to. As such, there exist two alternatives for the semantics:

1. *Shallow copy*, also called *one time binding*. To describe this alternative, let us assume that we have a declaration of a class `C` implementing an interface `I`. Then, every method introduced in `I` not explicitly implemented in `C` is implicitly implemented with the call to the method with the same name, compatible with the interface and introduced most closely to `C` in its hierarchy. However, those two method introductions are still distinct. Hence, every other implementation (in a subclass or in an applied mixin) will re-implement only one of these two methods introductions, either the method introduced in the interface, or the one introduced in the ancestors of `C`. The meaning of class `C` is equivalent to a class in which for every method `I.mi()` introduced in the interface, not implemented in `C`, but having compatible implementation in some `Cj` ancestor of `C`, we have the following implementation:

```
implement retType packageI.I.mi(...)
{ return this.packageCj.Cj.mi(...); }
```

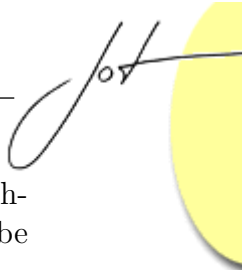
2. *Deep copy*, or *deep binding*. Let us take the example of the previous point. In this alternative, in class `C` and all its subclasses the method `I.mi` and `Cj.mi` will have the same implementation. Every re-implementation of those methods of the shape `implement retType I.mi` or `implement retType Cj.mi` will have the identical effect, that is, they will modify the implementation of both of them.

We decided to present as the official semantics the one of Section 6 because of its cleanness. We will only refer to the shallow copy alternative while defining the translation from Java to HygJava presented in Section 7.

## Notes on the type system

The type system of HygJava is a straightforward modification of the one of Java. The typing rules for the new constructs are:

- The HygJava method introduction is type-checked as a declaration of a Java abstract method. The most important check is whether a method introduction is not a declaration of a new method with the same name but with a different



result type. However, in HygJava, we must only check this against methods declared in the same module (while in Java the whole hierarchy must be checked).

- For every method implementation, we must check whether the referenced method introduction exists in the referenced module with the correct parameters and result type.
- The rule for checking the type of a method call is simpler because the method lookup is simpler. The compiler always knows the point of introduction, so it does not have to search the whole hierarchy.

Note also that using the hygienic methods approach has one, rather profound, consequence: in hygienic languages, the notions of nominal and structural types coincides. This happens because all hygienic method introductions are unique. In those cases in which fields occur in the public interface, the statement is still valid if we apply the same hygienic methodology to fields, see Section 8. This might lead to interesting theoretical consequences and we will address these as future work.

## 7 BACKWARD COMPATIBILITY

HygJava is, as we believe, a good proposal for a new language designed to solve ambiguity problems. However, because of the fact that HygJava is not a superset of Java, a hygienic compiler will not be able to compile most of the existing Java code. Therefore, it is necessary to develop a backward-compatible version of HygJava that integrate properly with existing working Java code, which might be non-hygienic.

To define the translation from Java to HygJava, we need three definitions.

**Definition 1** We define an IHierarchy of a module  $M$  as the smallest set such that:

- it contains module  $M$ ;
- for any class  $C$  in this set, it also contains all parent classes of  $C$  and all interfaces implemented by  $C$ ;
- for any interface, it contains all its parent interfaces.

**Definition 2** We define a set of introduction places of method  $m$  from module  $M$  (denoted  $IP_m^M$ ), as the biggest subset of IHierarchy of  $M$  such that:

- every element of  $IP_m^M$  contains a declaration of  $m$ ;
- for every  $M'$  in  $IP_m^M$ ,  $M'$  is the only element of the IHierarchy of  $M'$  containing a declaration of  $m$ .

**Definition 3** We define a primary introduction place of method  $m$  in module  $M$  (denoted  $PIP_m^M$ ) as the following element of  $IP_m^M$ :

- if the set  $IP_m^M$  contains a class (notice that it can contain at most one class) then  $PIP_m^M$  is that class;

- otherwise, if the  $M$  is a class, then  $PIP_m^M$  is an interface  $I$  from  $IP_m^M$ , such that the ancestor of  $M$ , which is implementing  $I$ , is the closest to the root (*Object*). If there is more than one such interface, then we choose any of them.
- If  $M$  is an interface, then  $PIP_m^M$  is any element of  $IP_m^M$ .

Then the backward-compatible translation from Java to HygJava is defined as follows:

- Any declaration of a method in an interface is interpreted as a method introduction.
- For any declaration method  $m(\dots)$  in class  $C$ , we take  $M$  as  $PIP_m^C$ . Then, if  $M=C$ , this declaration is interpreted as a method introduction together with its implementation (the implementation is omitted in the case of an abstract method). Otherwise, it is interpreted as a implementation of a method introduced in  $M$ .
- In every call to a method  $m$  on an object of type  $M$ , we prefix the name  $m$  with the path of  $PIP_m^M$ .
- For every class  $C$  implementing some interface  $I$  (or some sub-interface of  $I$ ), and every method  $m$  declared in  $I$ , let us take  $M$  as the  $PIP_m^C$ . If  $M$  is not  $I$ , then we add in  $C$  the following declaration (analogously to the shallow copy variant of Section 6):

```
implement retType package_I.I.m(...)
{ return this.package_M.M.m(...); }
```

This translation can be used in two ways:

- It can be incorporated in a HygJava compiler. With such an approach, the Java code can be left in its original form, and also new code can be developed in the classical Java style. Moreover, such code can also reference HygJava classes. However, when the source Java code is modified, the lack of hygiene may still cause problems as the ones discussed in Section 2.
- It can be supplied as a separate translator. This tool can be executed to translate Java code into hygienic one. The result then can be compiled directly with a hygienic compiler, therefore further modifications of the “hygienized” code will not cause any of the problems discussed in Section 2.

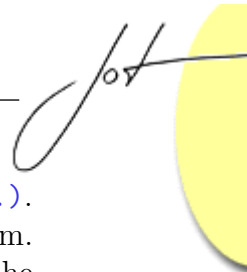
## 8 FINAL REMARKS

### Practical issues

The hygienic syntax is cleaner than the classical one and discards ambiguities, however it might look less convenient for practical usage.

This inconvenience can be easily solved by developing an appropriate IDE<sup>2</sup> for hygienic languages. Such an IDE will contain the following features:

<sup>2</sup>Integrated Development Environment.



- Method call expressions can be written in a classical style: `obj.m1(...)`. When a call is not ambiguous it is automatically expanded to its full form. Such an expansion is also performed in ambiguous cases, by choosing the introduction of `m1` from the module which is the closest in the hierarchy to the static type of `obj`.  
This behavior is similar to the way the Java compiler works, however the choice is done only once, therefore the code will not start working differently with newer versions of the libraries.
- Expressions can be normally displayed in the abbreviated form `obj.m1()`, while the full form can be invoked by clicking on the dot before the method name. The source code will always contain the full form.  
The hiding of some parts of the code is also present in most of the recent IDEs for languages like Java and C#.

However, it is important to notice that no IDE support alone can solve our problem. An IDE can be used to reduce the programming burden by highlighting when a new method declaration overrides an existing method in the code being written currently, but it will not protect from accidental ambiguities if the code will be used with other versions of the libraries it exploits. Such changes of libraries can occur, for example, when a core part of a software is sold to a series of customers, customized on the customer site, and updated periodically.

### Further extensions: fields, nested and anonymous classes

As already mentioned, ambiguity problems concerning methods can also occur with regard to fields. Those problems are caused by the fact that any code accessing a field occurring in the client code, or in a subclass of a class declaring this field, uses the name to reference it. This implies that declaring a new field with the same name in a subclass (or in a mixin) may make the existing code behave differently.

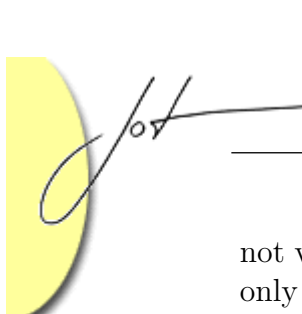
Therefore, to solve those problems, we can apply the same methodology as for methods. We can define *field introduction* with the same syntax as the Java field declaration, but additionally we can bind the identifier of a field to the place of declaration. Analogously, the syntax for field de-reference can be extended with the path to the place of the field introduction (the same way it is done for method calls).

The other issues (semantics, backward compatible interpretation, type checks and IDE support) follow automatically.

Similarly, the idea can also be extended to nested and anonymous classes. With nested classes, the only difference is that the name of the class is not unique globally, so it should be accompanied with the name of the enclosing class. In fact, this can be done the same way the compiler builds names for those classes, that is, we can use the following form of identifiers: `p1.PublicClass$NestedClass.m1()`.

In the case of anonymous classes, we do not have any distinctive unique identifier representing such a class. However, the type induced by such an anonymous class is





not visible from the outside, so all the identifiers introduced by it can be referenced only inside this class. Therefore, we can introduce a special keyword `current`, representing the local context, which can be used for referencing the identifiers introduced locally. That keyword can be used in method implementations: `implement String current.m1()`, and method calls: `obj.current.m1()`. Moreover, the same keyword may be exploited as a shortcut for paths in any class.

## Areas of application of our approach

In this paper we tackled problems concerning accidental name clashes in the Java language (and its possible mixin extensions), and presented a solution for them. While in Java-like languages the lack of hygiene can cause problems during the execution (when accidental overrides occurs), in a language like Eiffel, [12], in which the non-ambiguity of names is especially emphasized (the Eiffel compiler verifies if there are accidental overrides, and renaming of methods is enforced), a similar lack of hygiene can only cause errors during the compilation of the client code, which we believe it is better than having problems at run time.

Name-clash errors in Eiffel can be fixed by applying some modifications in the code, like the renaming of inherited methods to get rid of the conflicts. However, our hygienic approach applied to Eiffel would ensure, for example, that even clashes at compile time will never occur, for example after downloading a newer version of a base library.

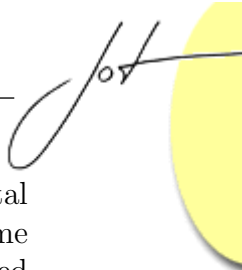
Similarly, in an aspect-oriented extension of Java called AspectJ, [11, 1], any name clash between class methods and aspect methods, and between aspect methods themselves, result in conflicts during the compilation. Unfortunately, when a new version of a library containing new methods that create conflicts is downloaded, the only way to make things compile again is to rename the methods in the client aspects and change all references to them (which can be a source of errors). Therefore, once again our hygienic approach would protect AspectJ from those problems, thus making it resilient to versioning.

## Related work

Some of the problems which we deal with in this paper have already been tackled. As an effect, in many practical languages and theoretical calculi, some mechanisms have been implemented to solve those problems at least partially.

Those solutions include the following ones:

- In the Delphi language, [2], which is the most popular implementation of Object Pascal, there is a direct distinction between the method implementation which introduces a new method and the one that redefines a method (via the keyword `override`). Nevertheless, all the references to methods are by name, therefore ambiguities can still occur.



- The designers of C#, [10], already saw the problems of possible accidental conflicts between different versions of the libraries. Therefore, in C# some features were implemented to address some of the problems we are concerned with.

First of all, C# allows one to distinguish an overriding method implementation from an introducing method implementation, via the use of the keywords `new` and `override`. However, similarly to Delphi, it allows the programmer to have more than one introduction, therefore a method implementation can still `override` a method introduction different from the one intended.

Additionally, C# distinguishes instance methods between `virtual` ones (dynamically dispatched ones) and statically dispatched ones, and as default behavior it chooses the statically dispatched ones. Notice that, in Java, a method can be overridden unless it is marked `final`, and even if a method is marked with this keyword, there is no possibility of declaring another method with the same method in a subclass. The approach chosen by the C#'s designers is that most of the methods cannot be overridden, therefore for methods not intended to be overridden at all, accidental overriding cannot occur. However, for `virtual` methods we can still have a problem: when introducing a method for the second time in a subclass, the implementation intended to `override` the first one now redefines the second one. Also, a method call expression can still suffer from ambiguous binding.

Finally, in a class implementing a method introduced in an interface, a programmer may declare explicitly from which interface this method comes (which is useful, when a method of the same name is declared in two different interfaces). The syntax is, in some respects, similar to ours:

```
interface I { void met(); };
interface J { void met(); };
class C : I, J
{ void I.met() {...}
  void J.met() {...}
}
```

However, this is only possible for methods introduced in interfaces and only with respect to the implementation of the method (not with respect to the method call), therefore it solves only some of the problems. Additionally, it has also some awkward behavior: the method `met()` cannot be executed on objects of type `C` without casting on the interface.

- The Eiffel language, [12], features some of the above described mechanisms.

First of all, a distinction in the syntax between method introduction and override also exists here, via the usage of the keyword `redefines`. However, while Delphi and C# allows one to have a few distinct introductions of a method with the same name, Eiffel raises an error when a new introduction of a method with an old name is found.

Additionally, Eiffel allows one to supply a distinct implementations for different methods with the same name inherited from different abstract classes (which play the role of interfaces in Eiffel). This can be achieved via the `rename` operation on the methods coming from different ancestor classes (notice also that Eiffel supports multiple inheritance) and the subsequent redefinition of each of the renamed methods.

- In C++, in particular in the presence of templates, the problem of non-hygienic identifier binding was pointed out by Smaragdakis and Batory in [16]. The solution proposed to solve ambiguities during method calls was to use the prefixing of the method name with the class name all the time (which is a feature of C++: `<class>::<method>(...)`). However, the hygienic programming is not enforced by the language, and, additionally, problems with ambiguities concerning the override are not addressed by this solution.
- Schärli et al., in their work on traits, [15], have also tackled the problem of accidental override. In order to solve this problem, they decided to: (i) not accept trait composition when accidental clashes between two traits used to build a class occur; and (ii) allow manual renaming of methods coming from traits. However, this approach requires manual modifications of different parts of the code. Additionally, a method implementation coming from a trait can still override accidentally one present in a super-class.
- In the MixedJava language, [8], the problem of having multiple implementations of methods with same name (but coming from separate mixins) is dealt with the concept of *view* of an object. Let us see this in the following example:

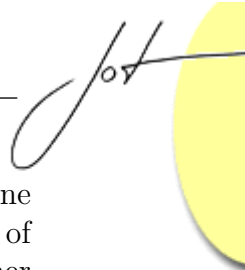
```

mixin M {void met() {...} };
mixin N {void met() {...} };
class A = M(N(Object));
...
M a = new A(); //type M indicates a view
N b = new A(); //N indicates another view
a.met(); //met() from M is called
b.met(); //met() from N is called

```

However, in a context where both methods are visible we still have a problem, as the chosen method might not be the one we expect.

- In the work on first class genericity for Java by Eric Allen et al., [3], a mixin is implemented by a generic class using its parameter as its ancestor. They introduced the notion of “hygienic mixin” to describe the semantics introduced by Flatt et al. and adapted it successfully to the world of generics. In contrast to MixedJava, MixGen has a compiler generating JVM-compatible bytecode, which uses the fully fledged name of the class in which a method is introduced to prefix the method name itself.



However, this prefixing is not visible in the source code because it is done implicitly during each compilation and class-loading, therefore the binding of methods in some class may change accidentally after modifications in other classes.

- The study on the “fragile base-class problem” by L. Mikhajlov and E. Sekerinski, [13], shows many different problems which can occur in unknown descendant classes, following the modification of an heir class. However, those problems are “semantical clashes” (concerning accidental incompatibility of behavior of modified methods), while in this paper we tackle “syntactical clashes” (concerning accidental compatibility of declarations of added methods).

## Conclusions

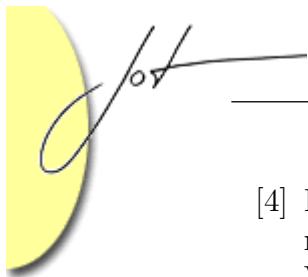
In our opinion, none of the solutions presented above solve the problem completely as our solution instead does. What our methodology offers can be summarized in one sentence: introducing new method identifiers and new fields in an existing class (or in an existing mixin) implementing some new functionality will never change the behavior of existing code (except, of course, for code using reflection mechanisms for finding methods by their names, which avoid statical verification). This might be seen as a special case of the general *Flexibility Theorem*, formalized in [13].

This result is achieved with a rather simple mechanism, i.e., an explicit prefixing, therefore we believe that our approach may be applied in the development of future languages and also (using backward compatibility) in new versions of existing languages. Having more and more components from different vendors of different versions makes it important to develop mechanisms which decrease the chances of inter-component incompatibility problems.

**Acknowledgments.** The authors would like to thank Gary T. Leavens and the anonymous referees for helping to improve this paper. We would also like to thank Pawel Urzyczyn for inviting the second author to Warsaw, and Marcin Kowalczyk for many fruitful discussions on object-oriented languages.

## REFERENCES

- [1] AspectJ Documentation. <http://www.eclipse.org/aspectj/docs.php>.
- [2] *Delphi Language Guide*. Borland Software Corporation, 2004.
- [3] E. Allen, J. Bannet, and R. Cartwright. A first-class approach to genericity. In *Proc. OOPSLA '03*, pages 96–114. ACM Press, 2003.



- [4] D. Ancona, G. Lagorio, and E. Zucca. Jam – a smooth extension of Java with mixins. In *Proc. ECOOP '00*, volume 1850 of *LNCS*, pages 145–178. Springer-Verlag, 2000.
- [5] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, The University of Utah, 1992.
- [6] G. Bracha and W. Cook. Mixin-based Inheritance. In *Proc. OOPSLA '90*, pages 303–311. ACM Press, 1990.
- [7] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, 2005.
- [8] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. In *Proc. POPL '98*, pages 171–183. ACM Press, 1998.
- [9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java<sup>TM</sup> Language Specification*. Addison-Wesley, Sun Microsystems, 2005.
- [10] A. Hejlsberg, P. Golde, and S. Wiltamuth. *C# language specification*. Addison-Wesley, 2003.
- [11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with aspectj. *Commun. ACM*, 44(10):59–65, 2001.
- [12] B. Meyer. An Eiffel Tutorial. Technical report, ISE Technical Report TR-EI-66/TU, 2001.
- [13] L. Mihajlov and E. Sekerinski. A Study of the Fragile Base Class Problem. In *Proc. ECOOP '98*, volume 1445 of *LNCS*, pages 355–382. Springer-Verlag, 1998.
- [14] D. A. Moon. Object-Oriented Programming with Flavors. In *Proc. OOPSLA '86*, pages 1–8. ACM Press, 1986.
- [15] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable Units of Behaviour. In *Proc. ECOOP '03*, volume 2743 of *LNCS*, pages 248–274. Springer-Verlag, 2003.
- [16] Y. Smaragdakis and D. S. Batory. Mixin-Based Programming in C++. In *Proc. GCSE '00*, volume 2177 of *LNCS*, pages 163–177. Springer-Verlag, 2001.



## ABOUT THE AUTHORS



**Jarosław Dominik Mateusz Kuśmierek** is a PhD student in Computer Science at the University of Warsaw, Poland. His main research interests are the base concepts and the design of object oriented languages, and the component oriented approach to software development. He can be reached at [jdk@duch.mimuw.edu.pl](mailto:jdk@duch.mimuw.edu.pl).



**Viviana Bono** is an Associate Professor in Computer Science at the University of Torino, Italy. Her main research interests are theoretical foundations, semantics, and design of object-oriented and functional languages. She can be reached at [bono@di.unito.it](mailto:bono@di.unito.it) and at <http://www.di.unito.it/~bono>.