Dynamic object extension for the Java virtual machine

Tom Tourwé* and Wolfgang De Meuter
{Tom.Tourwe,wdmeuter}@vub.ac.be
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
1050-Brussel-Belgium

Abstract

In this paper, we report on the experiences we had and the problems we encountered integrating dynamic object extension in Java. At first sight, this seems impossible due to the incompatibilities between the class-based and the prototype-based paradigms. Even more, because the Java virtual machine is tightly coupled to a class-based language, our goal seems impossible. We show, however, that dynamic object extension can be incorporated into a class-based language and compiled to the Java virtual machine, although some compromises need to be made.

1 Introduction

The Java language and its underlying virtual machine [Sun95] are becoming extremly popular, thanks to properties as platform independence, portability and automatic memory management. The virtual machine is well on its way to become universal hardware, as many companies are incorporating it in their products. This raises the question whether other languages than Java can benefit from this popularity and can be compiled to the instruction set of this machine. Of particular interest are prototype-based languages, since these incorporate some features not found in their class-based counterparts. In particular, the ability to extend objects at runtime seems incompatible with a virtual machine which is tightly bound to a class-based language. Dynamic object extension requires objects to be reentrant entities, which is not the case in a class-based language.

One form of dynamic object extension, however, seems to alleviate this problem somewhat. *Mixin methods*, first introduced in Agora [DM96], allow objects to be extended, but only in a predefined manner: the programmer has to specify statically (at compile-time) which extensions can take place at runtime. This distinguishing feature allows us to incorporate mixin-method based object extension in Java.

*Author financed with a doctoral grant from the *Instituut voor Wetenschap en Technologie*, Flanders

2 A short introduction to mixin methods

Mixin methods were introduced in order to overcome some problems with mixin-based inheritance [BC90]. Ordinary mixins can be applied to any class or object, even when the result of this application does not make much sense. For example, a `ColorMixin` mixin can be applied to a `Point` object, but also to a `Person` object, although an orange person clearly doesn't make any sense. Also, mixins can breach the encapsulation of an object, since the object cannot possibly foresee how it will be extended. For an example and a thorough discussion of this problem, we refer the reader to [DMMS96].

These problems do not exist when using mixin methods. The idea is that the programmer defines all mixins that can be applied to an object inside this object. Applying a mixin then boils down to sending the appropriate message to this object. The object will then look up the method that corresponds to the message and execute its body. In our example above, this means that the `ColorMixin` mixin should be defined inside of the `Point` objects, and that sending the appropriate message, like for example `makeColored`, to a `Point` object will apply the mixin.

The result of sending a mixin message to an object depends on whether the corresponding mixin method is defined as *functional* or as *imperative*. In the latter case, the receiver is destructively extended, while in the former case, an extended object is returned which has the receiver of the message as parent object.

Applying mixins via message passing has two advantages. First, it is no longer possible to apply a mixin to an arbitrary object: an object knows which mixins can be applied to it and it will fail to understand a message for which no mixin method was defined. Second, the encapsulation of an object cannot be breached as the corresponding methods should be defined beforehand by the programmer.

3 Mixin methods in Java

In this section, we will discuss the most important problems we ran into while integrating mixin methods in Java. After giving a general overview of the approach used, we'll explain why the method-lookup algorithm for statically-typed languages poses severe problems and how they can be solved. Second, we'll elaborate on late binding of self, which has different semantics in class- and prototype-based languages, and on how we tried to reconciliate these. Third, we'll discuss *delegating methods* and finally we'll describe the difficulties encountered with the inheritance of mixin methods.

3.1 General idea

Mixin methods define methods and instance variables that can be added to an object at runtime when sending it the appropriate mixin message. The definition of a mixin method thus shows close resemblance to a class in Java. Therefore, the representation best suited for the body of a mixin method is a class. Thus, for every mixin method defined in a class, the compiler generates

a so called *mixin class*, which contains all methods and variables defined in the mixin method. A mixin method can thus be seen as a special kind of inner class.

For example, consider the following class definition:

```
class A {
    <some methods and variables>
    mixin Object m() {
        <some methods and variables>
    }
}
```

The compiler generates two classes for this definition: the class `A` and the mixinclass `A$m`. When sending the mixin message `m` to an object of class `A`, an instance of the class `A$m` is returned.

Objects created by sending mixin messages need to have at least the same interface as their parent object, as they sometimes need to delegate certain messages to it. In a statically typed class-based language, a certain class has at least the same interface as another class if the former class is a subclass of the latter. Thus, to ensure compatible interfaces, we require mixin classes to be a subclass of the class in which the corresponding mixin method is defined. This means that, in our example, the mixin class `A$m` is a subclass of the class `A`.

3.2 Method-lookup strategy

Statically-typed object-oriented languages typically use a dispatch table for improving the performance of the method-lookup algorithm. This dispatch table can only be constructed when all the methods that a class defines are specified at compile time. Dynamic object extension however, allows methods to be added to objects at runtime. Thus, table-based dispatch poses some severe problems for incorporating dynamic object extension in class-based languages. When using mixin methods however, an object knows at compile time with which methods it can be extended at runtime. As a consequence, it is still possible to construct a dispatch table, since all methods are known statically.

A limitation inherently associated with this approach, however, is that only functional mixin methods can be used. As already explained, imperative mixin methods destructively change the receiver of the mixin message. If we wanted to integrate this feature into Java, this would mean we should be able to change the dispatch table of an object at runtime, which is not possible.

3.3 Late binding of self

In order to be able to support late binding of self in prototype-based languages, the receiver of a message and the self reference of the receiver must be able to differ. When a message is delegated from an object to its parent object, the self reference of the parent object should refer to the original receiver of the message, so self sends can be delegated to it. Stated otherwise: objects in prototype-based languages should be reentrant. In class-based languages, however, classes should be reentrant to support late binding of self, while objects can be fully closed entities [SDM95].

In order to simulate reentrant objects in class-based languages, we implicitely pass the receiver of a message as a first argument to this message. Then, instead of generating "normal" code for a self send, the compiler emits code that delegates the message to this extra argument. When messages are delegated to the parent object of the receiver, we do not pass the parent object as a first argument. Instead we pass the receiver of the original message. This ensures us that self sends occuring in the invoked method of the parent object will be executed on the original receiver, because they are delegated to this argument.

3.4 Delegating methods

Although methods of a class can be overriden in a mixin method, not all mixin methods override all methods of their enclosing class. However, because a mixin class is a subclass of its enclosing class, methods that are not overriden are inherited. This means that, when a message is sent, resulting in the execution of an inherited method, its body is executed on the wrong object! Rather than being executed on the receiver of the message, it should be executed on one of its parent objects.

As a solution to this problem, we let the compiler implicitly override each method that is not explicitely overridden by the programmer in the mixin method. The compiler-generated body of this method simply delegates the corresponding message to the parent object of the current receiver. This technique ensures us that the body of a method will always be execute on the right object.

3.5 Inheritance of mixin methods

Mixin methods, just like ordinary methods, can be inherited by subclasses. With the approach we have taken, however, this poses some problems. Take a look at the following example:

```
class A {
    ...
    mixin Object m() {
        ...
    }
}
```

Two classes are created by our compiler: the class `A` and the mixin class `A$m`. When the message `m` is sent to an instance of class `A`, an instance of class `A$m` is returned. Consider now the following subclass of class `A`, which does not override the mixin method `m`:

```
class B extends class A {
    ...
}
```

This class can also respond to the mixin message `m` since it is a subclass of class `A`. However, the result of sending this message is an object of class `A$m`, and these objects do not understand the messages added in class `B`! Therefore,

whenever a subclass is constructed whose superclass defines some mixin methods, these mixin methods need to be overriden in the subclass. This ensures that a new mixin class `B$m` is created, and that the mixin method in the subclass returns an instance of it. Of course, the compiler automatically takes care of this overriding of mixin methods, when the programmer does not explicitly do so.

4 Evaluation

Although it seems that we were able to solve some important problems, there still are some deficiencies we have to deal with. We will now discuss these in more detail.

In order to preserve the use of dispatch-table based method-lookup, we were obliged to exclude imperative mixin methods. These type of mixin methods have one important advantage however. Since they destructively change the receiver of a mixin message, they allow the programmer to change an object and all of its child objects in one stroke, just by sending a mixin message. This feature is no longer supported, as it is incompitable with the dispatch-table mechanism.

Since mixin methods can be inherited, a special form of inheritance, called *repeated inheritance* becomes possible. Repeated inheritance means that the programmer is allowed to extend an object by sending it the same mixin message over and over again. This feature allows the programmer to easily implement linked lists, for instance. Repeated inheritance is not supported by our approach, however, but we will not elaborate on this any further, as this would lead us too far.

A more serious problem we have to deal with is the typing problem. Since Java is a statically typed language, mixin methods should have types attached to them. However, it is generally known that statically typing dynamic object extension is difficult, if not impossible. Again, mixin methods can possibly alleviate this problem somewhat, because they are defined beforehand by the programmer. Some reseach on this topic has been conducted at our lab. For our research, however, we neglected this issue for the most part, and inserted the appropriate typecasts where needed. Of course, this often results in a loss of expressiveness.

5 Conclusion

In this paper, we showed that dynamic object extension can be integrated into a class-based language and compiled to the Java virtual machine, even though this machine is mainly targeted at class-based languages. To achieve this goal some compromises had to be made, however. First of all, mixin methods should be used as the extension technique, since these kind of methods need to be defined statically at compile time. Second, only the functional variant of mixin methods is allowed, since these do not destructively change the receiver. Together, these two properties allowed us to still use dispatch-table based method-lookup. Furthermore, we were able to solve some important problems: we showed how to mimick the reentrance of objects and the implicit delegation of messages

that prototype-based languages exhibit and we presented a technique to allow inheritance of mixin methods.

In the light of using prototype-based languages for distributed software systems, the results reported on in this paper can be of significant importance. Given the widespread use of the Java virtual machine and the fact that at least one prototype-based language can be compiled to it, it is definetely worth looking into how these languages can be used to implement distributed applications.

References

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. *Communications of the ACM*, 1990.

[DM96] Wolfgang De Meuter. *Agora96 Language Manual*, 1996.

[DMMS96] Wolfgang De Meuter, Tom Mens, and Patrick Steyaert. Agora: Reintroducing Safety in Prototype-based Languages. Technical Report vub-prog-tr-96-13, Programming Technology Lab, Vrije Universiteit Brussel, 1996. Presented at the ECOOP '96 Workshop on Prototype-Based Languages.

[SDM95] Patrick Steyaert and Wolfgang De Meuter. A Marriage of Class- and Object-Based Inheritance Without Unwanted Children. In *ECOOP '95 - Object-Oriented Programming*, Lecture Notes in Computer Science, pages 127–144. Springer-Verlag, 1995. Proceedings of the 9^{th} European Conference on Object-Oriented Programming. Aarhus, Denmark, August 1995.

[Sun95] Sun Microsystems. *The Java Virtual Machine Specification*, 1995.