Submission to the Workshop on Reflectively Extensible Programming Languages and Sytems (REPLS) at the International Conference on Generative Programming and Component Engineering (GPCE'03)

September 22nd, 2003. Erfurt, Germany

Efficient Support for Mixin-Based Inheritance Using Metaclasses

Noury Bouraqadi

bouraqadi@ensm-douai.fr
http://csl.ensm-douai.fr/noury
Ecole des Mines de Douai - France

Abstract

Mixin-based inheritance is an interesting alternative to both single and multiple inheritance. As opposite to single inheritance, it allows code reuse among different class hierarchies. Contrary to multiple inheritance, it allows developers explicitly specify the desired behavior by explicitly linearising superclasses. In this paper we show that reflection can support mixin-based inheritance without any performance overhead. This support is provided by means of *metaclasses* that are classes which instances are also classes.

1 Introduction

Code reuse is one of the holy grails of software engineers. It aims at avoiding code duplication and speeding up software production. So, when building new softwares, developers will ideally have to write only code corresponding to new features. The rest of the code is borrowed from previous developments.

In the context of object-oriented programming, inheritance is one major technique for reuse. Approaches to perform inheritance can be splited into two main families: single inheritance and multiple inheritance [Sin95]. None of those two families is totally satisfactory. Single inheritance does not allow code reuse among classes belonging to different hierarchies. And, multiple inheritance can lead to undesirable behavior because of complex linearisation algorithms for automatic conflicts resolution. In order to avoid these two limitations, one possible solution is to use *Mixin-based inheritance* [BC90]. A mixin is a subclass parametrized by its superclass. The superclass of a mixin varies according to the hierarchy where the mixin is used. Therefore, a mixin can be reused within different class hierarchies. Different mixins used in some hierarchy should be explicitly ordered by developers. This ordering can be viewed as an explicit linearisation that solve potential conflicts.

In this paper, we explore the implementation of mixins using reflection. As far as we know, prior to our work, only Brown et al described how to introduce mixin-based inheritance by means of reflection [BSK02]. Their solution relies on Java reflective facilities. More precisely, it heavily uses dynamic method invocation provided by the java.lang.reflect package. This approach has two main drawbacks. On the one hand, it forces developers use an unnatural method invocation protocol. Each method invocation requires writing several lines of codes to assemble arguments, wrap them if needed (i.e. primitive types), deal with any unhandled exceptions... On the other hand the use of dynamic method invocation introduces performance overhead.

In this article, we propose an efficient alternative to Brown et al work. Our research have been experimented using a reflective extension of Smalltalk named MetaclassTalk. Explicit *metaclasses* (i.e. classes which instances are also classes [Coi87]) are among reflective facilities provided by MetaclassTalk. We show that starting from a language supporting only single inheritance,

metaclasses can be used to introduce mixin-based inheritance without any performance overhead. Moreover, our approach allows mixin developers to use the default language constructs for method invocation.

In the following, section 2 presents the model of mixin-based inheritance we use. Section 3 describes the use of metaclasses to support mixin-based inheritance. Last, before the conclusion, section 4 sketches how to refactor classes into mixins.

2 Mixin-Based Inheritance

A mixin is a subclass parametrized by its superclass [BC90]. That is, a mixin can be reused within different class hierarchies. The superclass of a mixin varies according to the hierarchy where the mixin is used. The concept of mixin first appeared in CLOS [Kee89] as a programming style. It was introduce to avoid problems caused to multiple inheritance and the automatic linearisation of class hierarchies [Sin95]. Since mixins are not bound to any superclass, their order in a subclass definition corresponds to the linearisation.

2.1 Our Model for Mixin-Based Inheritance

In order to take benefit from mixins without having any drawbacks of multiple inheritance, we propose to somewhat constraint the CLOS model. We suggest to allow a given class inherit from an arbitrary number of mixins. But, as opposite to CLOS, a class should have at most one non-mixin direct superclass. Mixin direct superclasses always precede this latter in the linearisation list.

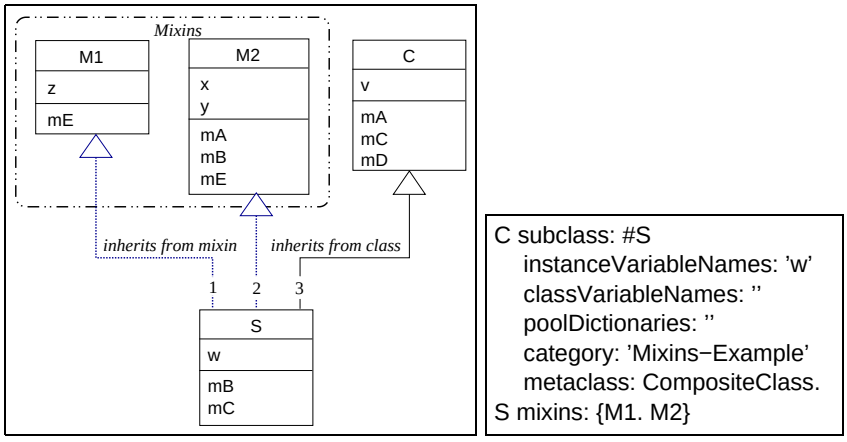

Figure 1: Example of a Class S inheriting from Two Mixins M1 and M2

We illustrate mixin-based inheritance using the following example. Suppose that we want to define the S subclass that inherit from the C non-mixin class and from mixins M1 and M2. The left most part of the figure 1 depicts the relationships among classes and mixins. We see that both classes (S and C) and mixins (M1 and M2) can hold fields and methods. While all fields should have different names, some methods defined either by mixins or by classes can have the same signature.

The inheritance arrows are numbered to denote the explicit linearisation. When an instance of S received a message, the method lookup starts at S. If required, the lookup continues at M1, then M2 and last C and its superclasses in order. Methods held by mixins can include super sends. Method lookup performed when evaluating such sends depends on the context of use of the mixin. In our example, suppose that M1 includes a method with a super message send. The evaluation of this message will lead to a method lookup starting at M2 and if necessary continuing through

C and its superclasses in order. We present more deeply lookup rules, linearisation and conflicts in section 2.2.

The code to build S is given -in the Smalltalk [GR83] syntax- by the right most part of figure 1. This definition provides mixins in the order they should appear in the linearisation list. This code also states (using the metaclass: keyword) that CompositeClass is the metaclass of S. This statement is important since it gives S the ability to inherit from mixins. More explanations about CompositeClass are provided in section 3.2.

2.2 Conflicts Resolution

Multiple inheritance leads to potential conflicts among superclasses and mixins. Conflicts can be of two kinds: fields conflicts and methods conflicts. Field conflicts are solved by forbidding the inheritance from two superclasses (mixins or not) which provide fields with same names. Method conflicts are solved using linearisation based on the following three rules. (*i*) Methods defined by a class precede methods defined by all its superclasses. (*ii*) Methods defined by a mixin direct superclass precede methods defined by a non-mixin direct superclass. (*iii*) And, within a list of mixins, methods of mixins that appear first precede methods defined by mixins appearing last. Since the list of mixins is explicitly provided when defining each subclass, linearisation is explicit. Indeed, developers have the ability to order mixins in such a way to get the desirable behavior.

Message selector	**Class holding the performed method**
mA	M2
mB	S
mC	S
mD	C
mE	M1

Table 1: Examples of Results of Method Lookups

In the definition of class S provided by figure 1, the M1 mixin appears before the M2 mixin. Thus, the result of linearisation is S, M1, M2, C. In case C has some superclasses (mixins or non-mixins) these latter would appear after C in the linearisation list. Table 1 gives examples of methods performed for different messages sent to an instance of S.

2.3 Dynamicity

In order to support run-time program adaptation we explored support for dynamic additions or removals of mixins in a subclass definition. Put another way, we need to support dynamic changes of the inheritance hierarchy. This latter is already possible in Smalltalk. Smalltalk does allow changing at run-time the superclass of a class. Instances of this latter are rebuilt to conform to the new definition of their class.

However, rebuilding instances is not enough for cleanly performing a change in an inheritance hierarchy. Also, initialization and clean up of resources introduced and used by mixins should be performed. As an example, resources accessible through new fields are not usually initialized[1]. And, values of removed fields are simply forgotten.

We introduce a new protocol to support smooth additions and removals for mixins. Every mixin holds two methods: mixinInitialize and mixinCleanUp that perform mixin specific initializations and clean up. The mixinInitialize method of a mixin M is evaluated for all instances of a class that start inheriting from M. The mixinCleanUp method of a mixin M is evaluated for all instances of a class that stop inheriting from M.

Initialization and clean up are not the only problems that arise when dynamically adding and removing mixins. Indeed, dynamicity also requires some mechanism that automatically deals

[1]Unless we use read accessors that perform field lazy initialization.

with conflicts that an addition of a mixin at run-time may raise. Explicit linearisation of mixins partially addresses this issue. However, by the time of writing, we did not yet explore deeply enough this issue and we defer it to another paper.

3 Metaclasses for Mixin-Based Inheritance

We implemented the mixin model described in section 2 within *MetaclassTalk*[2] a reflective extension of Smalltalk [BL02]. The current implementation of MetaclassTalk have been developed with the open source Smalltalk named Squeak [IKM+97][GR02]. MetaclassTalk extends Smalltalk in two main directions. First, and contrary to Smalltalk, MetaclassTalk provides explicit metaclasses[3]. Explicit metaclasses are handled (creation, instantiation) in the same way as are plain classes. Second, MetaclassTalk provides a MOP (Meta-Object Protocol [KdRB91]) that allow changing the language semantics (e.g. message dispatch, read/writes of fields). In the following, we focus on the metaclass support which is the only feature used for implementing mixin-based inheritance.

3.1 Implementation Through Class Generation

Conceptually, mixin-based inheritance introduced in section 2 is a kind of multiple inheritance. However, our implementation fully relies on single inheritance. The result of the linearisation corresponds to the actual inheritance hierarchy. The link between multiple inheritance and single inheritance is done by viewing mixins as subclass builders, as suggested by Bracha et al. [BG96]. A mixin takes a class as input and produces a subclass of the given class. The new subclass will include fields and methods which definitions are held by the mixin.

Subclasses built by mixins are not directly available to developers. They are implicit. Then, one can not refer to them directly. Instead, developers only deal with mixins.

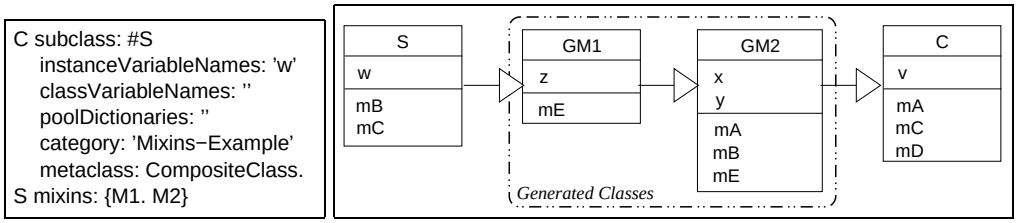

Figure 2: Definition of a Class Inheriting form Two Mixins and the Resulting Inheritance Hierarchy

Figure 2 gives the actual inheritance hierarchy of a class S inheriting from a non-mixin superclass C and from two mixins M1 and M2. Each mixin generates a new implicit class that is inserted between S and C. In figure 2 names of generated classes are prefixed with "G". The order in which mixins are listed in the definition of S is important. It gives the ordering of generated classes into the inheritance hierarchy. In our example, the M1 mixin appears before the M2 mixin in the definition of S. Then, the GM1 class generated by M1 is the superclass of the GM2 class generated by M2.

3.2 Explicit Metaclasses Used for Implementation

We implemented mixin-based inheritance using three explicit metaclasses: Mixin, CompositeClass, and GeneratedClass.

[2]MetaclassTalk can be downloaded at http://csl.ensm-douai.fr/MetaclassTalk

[3]Smalltalk metaclasses are implicit: they are anonymous and automatically handled by the system.

Mixin: Because we view mixins as a special kind of classes, we describe them using the Mixin metaclass. We found that this choice eases the use of mixins since we can reuse all tools available for classes (Browsers, senders/implementors of methods, ...). So, using a browser, one can define a mixin within some category, comment the mixin or implement the mixin's methods. mixins hold definitions of fields and method to copy into generated classes. They also holds references on generated classes to update them whenever a change occurs (e.g. on methods additions or removals).

CompositeClass: A composite class is a class that conceptually can have many superclasses. It is the responsibility of a composite class to enforce mixin-based inheritance rules. It forbids the inheritance from more than one non-mixin superclass or from two superclasses that hold homonym instance variables. The inheritance from mixins is materialized as a protocol for adding and removing mixins. When adding or removing mixins, the composite inserts or removes from its actual inheritance hierarchy classes generated by mixins.

GeneratedClass: A generated class is an implicit class built by some mixin. It has the responsibility of computing its *class format*. The class format is used by the Smalltalk virtual machine to determine the number of memory bytes to allocate for a new object. The computation of a class format takes into account the number of all instance variables of a class, including inherited ones. Each generated class has the responsibility to recompute its class format whenever its structure changes (e.g. removal of a superclass, addition of an instance variable,...). Generated classes have also the responsibility to hold copies of methods of the mixin responsible of the generation. Although it results in some space overhead, the decision to make copies of methods has important benefits for efficiency. Since methods can include messages sent to super, we need to compute the context of the method in order to perform the dispatch of such messages. By copying methods and because generated classes are arranged in a single inheritance tree, this computation can be done at compile time. Moreover, we can rely on the default mechanism for method lookup provided by the Smalltalk virtual machine. Therefore, the use of mixin-based inheritance does not alter the execution performance.

3.3 A Word About Performance

Our implementation maps the "conceptual" multiple inheritance from mixins into a single inheritance hierarchy. Therefore, the method lookup provided by the Smalltalk virtual machine is used and no overhead is introduced.

The use of our implementation only introduces a necessary duplication of methods. This is because methods held by mixins can include messages sent to super. Dispatching those messages requires taking into account hierarchies where the mixin is used. For a same message, the class where to start lookup can be different for different hierarchies. However, for a given mixin, each generated class participates to a single hierarchy. So, the class where to start lookup for such messages can be obtained at compile-time from generated classes. This is why, methods defined by mixins are copied and compiled into generated classes.

4 Extracting Mixins From Classes

For the purpose of reuse or refactoring one may need to extract mixins from some existing class. By extracting mixins we mean building one or more mixins based on some given class. Those mixins will hold copies of fields and methods provided by the given class. On extraction, this latter class can be: (i) left unchanged, (ii) refactored to use generated mixins, (iii) transformed into a mixin, or (iv) deleted.

An extraction of mixins without changing the original class is straight forward. Instances of the Mixin metaclass are built by copying methods and fields from the original class.

In the case where the the original class is refactored, we need first to build mixins by copying the right fields and methods. Then, these elements should be deleted from the original class. Next, our class should become instance[4] of the CompositeClass metaclass in order to be able to inherit from mixins. Last, the resulting class should inherit from mixins extracted in the first step.

Transforming a class into a single mixin can be done by making the class become instance of the Mixin metaclass. An important issue is related to the inheritance tree of the original class. After the transformation, references to the superclass will be lost. So, messages sent to super if any, will be resolved according to the context where the mixin is applied. Besides, in the case where the transformed class has one or more subclasses, these latters should be refactored as following. Suppose that we want to transform a class named Y into a mixin. Also, suppose that Y inherits from a class Z and has two subclasses X1 and X2. Before transforming Y into a new mixin, we need to first to make X1 and X2 become subclasses of Z. Next, X1 and X2 should become instances of CompositeClass. Last, X1 and X2 should inherit from the new mixin built out of Y.

5 Conclusion

In this paper we presented an approach to implement mixins using reflection. This implementation was performed using MetaclassTalk a reflective extension of Smalltalk. MetaclassTalk introduces different reflective facilities including explicit metaclasses. We showed that only explicit metaclasses are necessary to support mixin-based inheritance.

Our approach consists in using metaclasses to generate a single inheritance hierarchy based on classes inheriting from multiple mixins. As a result, we can take benefit of method lookup provided by the virtual machine. Thus method lookup performance is exactly the same whether we use mixins or not.

References

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proceedings of ECOOP/OOPSLA'90*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[BG96] Gilad Bracha and David Griswold. Extending smalltalk with mixins. OOPSLA'96 workshop on Extending Smalltalk, October 1996.

[BL02] N. Bouraqadi and T. Ledoux. Aspect-oriented programming using reflection. Technical Report 2002-10-3, Ecole des Mines de Douai, October 2002.

[BSK02] T.J. Brown, I. Spence, and P. Kilpatrick. Mixin programming in java with reflection and dynamic invocation. In *Proceeding of the Conference on Principles and Practice of Programming in JavaT*, Kilkenny City, Ireland, June 2002.

[Coi87] Pierre Cointe. Metaclasses are First Class Objects: the ObjVLisp Model. In *Proceedings of OOPSLA'87*, pages 156–167, Orlando, Florida, USA, October 1987. ACM.

[GR83] Adele Goldberg and David Robson. *Smalltalk 80*, volume The Language and its implementation. Addison-Wesley, 1983.

[GR02] M. Guzdial and K. Rose, editors. *Squeak: Open Personal Computing and Multimedia*. Prentice Hall, 2002.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Allan Kay. Back to the Future. The Story of Squeak, A Practical Smalltalk Written in Itself. In *Proceedings of OOPSLA'97*, pages 318–326, Atlanta, Georgia, October 1997. ACM.

[KdRB91] Gregor Kiczales, Jim des Rivires, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kee89] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, Reading, Massachusetts, USA, 1989.

[Sin95] Ghan Bir Singh. Single Versus Multiple Inheritance in Object Oriented Programming. *OOPS Messenger*, 6(1):30–39, January 1995.

[4]To perform the metaclass change, all classes in MetaclassTalk understand the changeMetaclass: message.