Safe Metaclass Composition Using Mixin-Based Inheritance

Noury Bouraqadi

bouraqadi@ensm-douai.fr
http://csl.ensm-douai.fr/noury
Dépt. G.I.P. - Ecole des Mines de Douai
941, rue Charles Bourseul - B.P. 838
59508 Douai Cedex - France

Abstract

In the context of meta-programming and reflective languages, classes are treated as full fledged objects which are instances of other classes named *metaclasses*. Metaclasses have proved to be useful for defining new class properties. Examples of such properties are lazy memory allocation, multiple inheritance, having a single instance... A class with some property is obtained by instantiating a metaclass which implements the desired property. However, instantiation allows assigning to a class only properties defined by a single metaclass. A composition mechanism is needed to reuse properties defined by different metaclasses and assign them to a given class. This composition should be performed without breaking *class-metaclass compatibility*. The compatibility issue arises when a class is coupled to its metaclass. So, when composing metaclasses, we need to take care of such coupling to avoid run-time exceptions.

In this paper, we explore the use of *mixin-based inheritance* to perform metaclass composition. Mixin-based inheritance is an interesting alternative to both single and multiple inheritance. As opposite to single inheritance, it allows reusing code among different class hierarchies. Contrary to multiple inheritance, it allows developers to explicitly specify the desired behavior through explicit linearizations. Our proposal is to define and compose reusable class properties by introducing mixins at the metaclass level. We demonstrate that this introduction can be done efficiently without altering compatibility.

Key words: Mixin, Metaclass, Class Property, Reuse, Composition, Compatibility

1 Compatibility and Metaclass Composition

Metaclasses (*i.e.*, classes which instances are also classes) have proved to be useful [1][2][3][4][5][6][7]. One of their main advantages is their use for the definition of new *class properties*. Where a class property represents a class specific behavior such as being abstract, having a unique instance, following a specific inheritance schema, etc...

Developers can make use of explicit metaclasses to build new kinds of classes. However, they should be aware of the *compatibility* issue [8][9]. This problem occurs when there is an implicit coupling of a class and its metaclass. To understand this coupling consider the left most drawing of Figure 1. The A class provides an instance method i-foo that sends some message c-bar to the class of the receiver. Put another way, the i-foo instance method makes the assumption that the class of the receiver understands the c-bar message. This assumption is true for A, since its metaclass AMeta implements a c-bar method. But, now look at B subclass of A. It inherits the i-foo instance method. To make i-foo still run smoothly we need to guarantee that B understands the c-bar message. So, to avoid a run-time exception (message not understood) we need to ensure that BMeta (*i.e.*, the metaclass of B) is compatible with AMeta and provides somehow a c-bar method.

A symmetric issue arises when a class method makes an assumption about instance behavior. An example illustrating this problem is given by the right most drawing of Figure 1. The AMeta metaclass implements a method c-foo that sends a message i-bar to a new instance. This method works fine for the A class (instance of AMeta) which provides the i-bar instance method. But, c-foo may lead to a run-time exception for B (instance of BMeta) which does not provide the i-bar instance method.

to ensure compatibility, metaclasses should be organized in such a way to comply with the model we introduced in [9] and which is depicted by Figure 2. This model has two layers of metaclasses: *compatibility metaclasses* and *property metaclasses*. Each class is the unique instance of a property metaclass. This metaclass holds class specific properties, *i.e.*, properties that are not propagated to subclasses. Class methods that are involved in class-metaclass couplings should be defined in compatibility metaclasses. These latters are organized in an inheritance hierarchy parallel to the class hierarchy.

Another issue that meta-programmers quickly face is the need to compose metaclasses. This composition is required in order to assign different properties to a given class. Assigning properties to classes is achieved by means of instantiation. A class with some property is obtained by instantiating the metaclass which provides the desired properties. But, instantiation is not enough when we want to reuse properties provided by different metaclasses. An extra mechanism is needed to compose (and hence reuse) existing metaclasses.

Since metaclasses are classes, they are composed using inheritance. Approaches to perform inheritance are split into two main families: single inheritance and multiple inheritance [10]. None of those two families is totally satisfactory. Single inheritance does not allow reusing code among metaclasses belonging to different hierarchies. And, multiple inheritance can lead to undesirable behavior because of complex linearization algorithms for automatic conflicts resolution. To avoid these two limitations, one possible solution is to use *Mixin-based inheritance* [11][12]. A mixin is a subclass parametrized by its unique superclass. The superclass of a mixin varies according to the hierarchy where the mixin is reused. Different mixins used in some hierarchy should be explicitly ordered by developers. This ordering can be viewed as an explicit linearization that helps solving possible conflicts among composed mixins.

In this article we explore the use of mixins to support metaclass composition. First, section 2 presents mixin-based inheritance. Then, section 3 shows that mixins can be used at the metaclass level to define reusable class properties. Section 4 describes the introduction of mixin-based inheritance within the compatibility model. Next, in section 5 multiple inheritance of mixins is used to compose metaclasses and hence assign different properties to a single class. In section 6, an overview of the three metaclasses used for implementing mixin-based inheritance is given. Section 7 provides a comparison with related works. Last, the paper ends with some concluding remarks, after sketching future works (section 8).

2 Mixin-Based Inheritance

According to Bracha and Cook [11], "*A mixin is an abstract subclass that may be used to specialize the behavior of a variety of parent classes*". It means that a mixin is a class parametrized by its superclass. Developers may choose to use a same mixin in unrelated inheritance hierarchies. So, the superclass of a mixin varies according to the hierarchy where the mixin is used.

We use mixins in the same way they are used in CLOS [13], *i.e.*, to achieve multiple inheritance. A class can inherit from different superclasses (mixins and plain classes). However, in CLOS mixin-based inheritance is just a programming style. We propose to use a constrainted model where only multiple inheritance of mixins is permitted. While a class can inherit from an arbitrary number of mixins, it should have at most one non-mixin superclass. Moreover, mixins direct superclasses precede non-mixins direct superclasses in the linearization list.

For example, consider the hierarchy depicted by Figure 3. As stated in the definition provided by the rightmost part of Figure 3, class B is a subclass of class A and inherits from mixins M1 and M2 in this order. Because of the provided mixins order, and because we privilege mixins over non-mixins, the result of linearizing B superclasses is:

{M1. M2. A}

This list gives the order of traversing superclasses of B on method lookup. It shows that methods defined by M2 override methods with the same selector [1] defined by A. And methods defined by M1 override methods of both M2 and A. Also, if there is a message sent to super within M2, method lookup will start at A. While lookup on a message sent to super within M1 will start at M2.

Now, look at class C. It inherits from the M3 mixin and from B. Linearization of superclasses of C leads to the following list:

{M3. B. M1. M2. A}

Note that result of linearization of superclasses of B is in the tail of this list. So, the intent of the implementor of B is not altered. Also, note that the order of superclasses in a linearization list is the one given by developers when defining subclasses. So, there is no possible ambiguity.

Last, we should point that the inheritance model we use forbids having two instances variables [2] with the same name. So, the creation of a subclass that inherits from two superclasses [3] that provide two homonym instance variables is forbidden.

3 Class Properties Reuse Using Mixins

In this section we show how to reuse class properties by means of mixins at the metaclass level. We illustrate this reuse with an example based on the Smalltalk Boolean class and its subclasses True and False. When studying the code of this hierarchy, one can identify at least two class properties: abstract and singleton. The Boolean class is abstract since it relies on its subclasses to provide concrete definitions for some methods. The two subclasses True and False are concrete. But, each of them should not have more than a single instance.

We implemented the above mentioned class properties using two mixins: Abstract and Singleton. We use this mixins to enforce properties of classes be-

[1] *i.e.*, signature.
[2] *i.e.*, fields.
[3] Two mixins, or a mixin and a non-mixin, or the same mixin twice.

longing to the Boolean hierarchy. This enforcement is achieved by making BooleanMeta, TrueMeta and FalseMeta inherit from the appropriate mixins (see Figure 4). It worth noting that, Singleton is reused twice as a superclass of both TrueMeta and FalseMeta. More generally, by implementing class properties using mixins, we can reuse them in unrelated metaclass hierarchies.

Figure 5 provides the code for the Abstract mixin that defines the "abstract" class property. Lines 1 to 3 build the mixin by instantiating the Mixin class. This code is what actually developers write when building the mixin. This is also what is shown within a browser when displaying the Abstract mixin definition. The new mixin holds no instance variables and defines the new method (lines 5 to 6). This latter raises an exception and hence forbids the creation of new instances.

Figure 6 provides the code for the Singleton mixin that defines the singleton class property. Lines 1 to 3 build the mixin and provide it with an instance variable necessary for the storage of the sole instance. Then, the mixin is provided a definition of the new method that raises an exception (lines 5 and 6). This is because the unique instance should be retrieved using the message uniqueInstance. The singleton instance variable is setup using the setupSingleton method (lines 12 to 15). This method creates a new instance only if no one is already available. Since Singleton is a mixin, its superclass varies according to the hierarchy where it is used. As a result, for a given message sent to super, the class where method lookup starts varies according to the hierarchy where the mixin is used. This is the case for the message new which is sent to super within the setupSingleton method.

4 Mixins and Class-Metaclass Compatibility

In the previous section, we didn't show relationships between metaclasses of the boolean hierarchy for sake of simplicity. However, as described in section 1, we must ensure compatibility by adhering to the model introduced in [9]. Figure 7 shows the boolean hierarchy refactored to comply with this model. For each class we have two metaclasses: a compatibility metaclass that ensures compatibility, and a property metaclass that provides class specific properties. Property metaclasses inherit both from compatibility metaclasses and from mixins that implement class properties.

The linearization lists of metaclasses of the Boolean hierarchy (see Figure 8) show that the introduction of mixin-based does alter compatibility. Indeed, the hierarchy of compatibility metaclasses remain parallel to the class one. In the linearization list for the metaclass hierarchy of True, TrueCompatibilityMeta appears right before BooleanCompatibilityMeta. Similarly, in the linearization of the metaclass hierarchy of False, FalseCompatibilityMeta appears right before BooleanCompatibilityMeta.

The use of mixins does still allow assigning specific properties to classes. Indeed, making property metaclasses inherit from mixins does not introduce any unwanted class properties propagation. In our example, the Abstract mixin appears only in the linearization list for the metaclass of Boolean. Thus, while the abstractness of Boolean is enforced, True and False remain concrete.

5 Class Properties Composition Using Mixins

Thanks to multiple inheritance of mixins, it is possible to assign many properties to a given class. To illustrate this idea, consider the True class. Besides having a unique instance, True is a *final* class. That is, True should not be subclassed.

The property of being final is implemented using a mixin we name Final. Making True both final and singleton is achieved by making TruePropertyMeta inherit from the two mixins Singleton and Final as shown in Figure 9. This is done by including the following expression into the definition of TruePropertyMeta:

TruePropertyMeta mixins: {Singleton. Final}

Based on our mixin-based inheritance model rules, the linearization list of direct superclasses of TruePropertyMeta is the following:

{Singleton. Final. TrueCompatibilityMeta}

Mixins appear in the order provided in the definition of TruePropertyMeta, and the non-mixin direct superclass appears after the mixin superclasses. In this example, metaclasses corresponding to the composed class properties (Singleton and Final) are orthogonal. So, no conflict arises when composing them. However, in case of conflicts, mixin-based inheritance rules apply. Two class properties which implementation (*i.e.*, the corresponding mixin metaclasses) make use of homonym instance variables can not be assigned to a same class. An attempt to perform a such assignment fails. But, one can assign properties which implementation provide methods with same selectors. Indeed, such conflicts are automatically solved using method overriding. Methods defined by a mixin are overridden by methods held by mixins appearing first in the definition a metaclass.

6 Implementation

We implemented the mixin model described in section 2 within *MetaclassTalk* [4] a reflective extension of Smalltalk [16][17]. MetaclassTalk extends Smalltalk in two main directions. First, MetaclassTalk provides explicit metaclasses [5]. The creation and the instantiation of explicit metaclasses are performed in the same way as for plain classes. Second, MetaclassTalk provides a MOP (Meta-Object Protocol [18]) that allows to change the language semantics (*e.g.*, message dispatch, read/writes of instance variables). In the following, we focus on the metaclass support which is the only feature used for implementing mixin-based inheritance.

6.1 Implementation Through Class Generation

Conceptually, mixin-based inheritance introduced in section 2 is a kind of multiple inheritance. However, the implementation fully relies on single inheritance. The result of the linearization 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.* [19]. A mixin takes a class as input and produces a subclass of the given class. The new subclass will include instance variables and methods which definitions are held by the mixin. A such subclass is implicit. It is not directly available to developers, and does not appear in class browsers. Then, developers only deal with mixins.

Figure 10 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 10 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 a subclass of the GM2 class generated by M2.

---

[4] The current implementation of MetaclassTalk have been developed with the open source Smalltalk named Squeak [14][15]. It can be downloaded at http://csl.ensm-douai.fr/MetaclassTalk

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

We implemented mixin-based inheritance using three explicit metaclasses:

- Mixin: describes mixins.
- CompositeClass: describes classes which inherit "conceptually" from multiple mixins.
- GeneratedClass: describes implicit classes built and maintained by mixins.

The metaclass 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. The set of instances of Mixin includes metaclasses that implement class properties such as Abstract or Singleton. Such (meta)classes hold definitions of instances variables and method to copy into generated classes. They also hold references on generated classes to update them whenever a change occurs (*e.g.*, on methods additions or removals).

The metaclass CompositeClass. A composite class is a class that conceptually can have many superclasses. This is the case of property metaclasses. 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.

The metaclass 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 is 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.

7 Related Works

Most programming languages providing metaclasses ensure compatibility without allowing class specific properties. This is the case of CLOS [13], Smalltalk [20], and SOM [3][7].

By default, CLOS ensures that each class and all its subclasses are instances of a same metaclass. While this constraint ensures compatibility, it forbids assigning specific properties to classes. However, CLOS allows changing this policy in order to use any metaclass. But, then no support is provided for compatibility.

As opposite to CLOS, Smalltalk allows assigning properties to classes while still ensuring compatibility. Indeed, the system ensures compatibility by organizing metaclasses into an inheritance hierarchy parallel to the class one. Although developers can not link a class to some specific metaclass, they can add instance variables and methods to metaclasses to define class properties. However, the parallel inheritance hierarchies lead to unwanted propagation of class properties. For example, subclasses of an abstract class will implicitly become abstract. Besides, single inheritance forbids reusing class properties across inheritance hierarchies.

The SOM system does not suffer from this latter limitation since it relies on multiple inheritance for reusing and composing class properties. Developers are allowed to select the M metaclass to use for creating S, a subclass of an existing C class. To ensure compatibility, SOM generates[6] a new metaclass, called *derived metaclass*, that inherits from M and from the metaclass of C. The new class S is created by instantiating the derived metaclass. This use of multiple inheritance allows reusing class properties in unrelated metaclass hierarchies. However, this solution propagates superclass properties to subclasses, and does not fully ensure compatibility [9].

NeoClassTalk has been the first language to support class properties reuse and composition, while ensuring compatibility [9]. Class properties are stored in the form of strings. Whenever a class property is needed, a metaclass is built from the appropriate string. This approach has two drawbacks. On the one hand, it is ad hoc. And on the other hand, definitions of class properties

[6] This generation is performed only when the M metaclass does not inherit from the metaclass of C.

are not compiled. Therefore, syntactic bugs (*e.g.*, missing period, parenthesis mismatch) can not be easily detected and fixed. Our solution based on the use of mixins avoids these two drawbacks. Mixin-based inheritance is a general purpose solution that works at both the class and the metaclass levels. And, since mixins are treated as full fledged classes, the code they provide is compiled before use.

A possible alternative to mixins is to use *traits* [21] at the metaclass level [22]. Roughly, a trait is an entity that holds a set of methods and allows reusing them in different class hierarchies. This exported behavior may be parametrized through a specified set of required methods. From the composition point of view, the trait model provides developers with a fine control over composition and conflict resolution. While mixin-based inheritance allows explicit ordering of mixins, traits composition goes down to the method level. Different operations (aliasing, exclusion,. . .) are possible for composing methods provided by different traits and hence solving conflicts. However, traits do not allow to reuse state, since they do not hold instance variables. While this characteristic eases composition, it also restricts reuse opportunities to method definitions. A class property which definition requires one or more instance variables can not be fully defined using traits. So, from this point of view mixins are superior to traits.

8 Future Works

A first important work we are currently conducting is to allow mixin composition even in case of conflicts due to instances variables. Such conflicts arise on repeated inheritance from a same mixin, or when different mixins provide instance variables with a same name. Our goal is to allow developers to decide about the appropriate action to perform. Candidate actions are:

- reject the composition,
- merge conflicting instance variables,
- accept duplication.

Besides improving instance variable composition, we aim at borrowing the interesting ideas provided by the traits model [21]. As we saw in section 7 the traits model is superior to mixins for the point of view of mixin composition. However, mixins allow state reuse (*i.e.*, instance variables) while this is not possible using traits. We are currently exploring a solution which merges advantages of both traits and mixins.

Another direction of investigation is to generalize the use of mixins. Instead of having both concepts of classes and mixins, we would like to only have mixins. Developers will only build and compose mixins. We plan to study the feasibility of this approach and its consequences on building large libraries. We plan to use Smalltalk for experimenting the generalization of mixin use. In this context, we need to deal with an extra issue: providing support for class variables, pool dictionaries.

Yet another work of high interest is the integration of mixins within the Smalltalk kernel. This integration will lead to the bootstrapping of our implementation. Indeed, our current implementation of mixin-based inheritance relies on three metaclasses. These latters are not mixins although they represent three different class properties.

9 Conclusion

In this paper, we presented an approach for defining reusable and composable class properties. Our solution consists in introducing mixin-based inheritance at the metaclass level. Class properties are defined as mixins that are reused in different metaclass hierarchies.

We also showed that mixin-based inheritance can be used at the metaclass level without altering compatibility. Our starting point was a model that takes care of possible couplings between classes and metaclasses [9]. We described how to make use of mixins within this model. Therefore, we can use mixin-based inheritance to assign specific properties to classes while still ensuring compatibility.

Experiments related to this research were conducted using MetaclassTalk a reflective extension of Smalltalk. This extension introduces both explicit metaclasses and extra reflective facilities. However, only explicit metaclasses were necessary to support mixin-based inheritance. The resulting implementation is very efficient: method lookup is equally fast with or without mixins.

Although we focused on metaclass composition and class properties reuse, our implementation of mixin-based inheritance is general purpose. It can be exploited for code reuse in any context and not only for metaclasses. We plan to apply it to other contexts and particularly for refactoring large class libraries.

Acknowledgements

The author thanks Stphane Ducasse, Houssam Fakih and Thomas Ledoux for their comments on a draft version of this paper.

References

[1] J.-P. Briot, P. Cointe, Programming with Explicit Metaclasses in Smalltalk, in: Proceedings of OOPSLA'89, ACM, New Orleans, Louisiana, USA, 1989, pp. 419–431.

[2] P. Cointe, The ClassTalk System: a Laboratory to Study Reflection in Smalltalk, in: Informal Proceedings of the First Workshop on Reflection and Meta-Level Architectures in Object-OrientedProgramming, OOPSLA/ECOOP'90, 1990.

[3] S. Danforth, I. R. Forman, Reflections on Metaclass Programming in SOM, in: Proceedings of OOPSLA'94, 1994, pp. 440–452.

[4] I. R. Forman, M. H. Conner, S. Danforth, L. K. Raper, Release-to-Release Binary Compatiblity in SOM, in: Proceedings of OOPSLA'95, 1995.

[5] T. Ledoux, P. Cointe, Explicit Metaclasses as a Tool for Improving the Design of Class Libraries, in: Proceedings of ISOTAS'96, LNCS 1049, Springer-Verlag, Kanazawa, Japan, 1996, pp. 38–55.

[6] M. N. Bouraqadi-Saâdani, T. Ledoux, F. Rivard, P. Cointe, Providing explicit metaclasses in smalltalk, in: OOPSLA'96 workshop : "Extending the Smalltalk Language", 1996.

[7] I. R. Forman, S. H. Danforth, Putting Metaclasses to Work, Addison Wesley, 1998.

[8] N. Graube, Metaclass Compatibility, in: Proceeding of OOPSLA'89, New Orleans, Louisiana, USA, 1989, pp. 305–315.

[9] N. Bouraqadi, T. Ledoux, F. Rivard, Safe Metaclass Programming, in: Proceedings of OOPSLA'98, ACM, 1998.

[10] G. B. Singh, Single Versus Multiple Inheritance in Object Oriented Programming, OOPS Messenger 6 (1) (1995) 30–39.

[11] G. Bracha, W. Cook, Mixin-based inheritance, in: N. Meyrowitz (Ed.), Proceedings of ECOOP/OOPSLA'90, ACM Press, Ottawa, Canada, 1990, pp. 303–311.

[12] M. Flatt, S. Krishnamurthi, M. Felleisen, Classes and mixins, in: Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, USA, 1998, pp. 171–183.

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

[14] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay, Back to the Future. The Story of Squeak, A Practical Smalltalk Written in Itself, in: Proceedings of OOPSLA'97, ACM, Atlanta, Georgia, 1997, pp. 318–326.

[15] M. Guzdial, K. Rose (Eds.), Squeak: Open Personal Computing and Multimedia, Prentice Hall, 2002.

[16] N. Bouraqadi, T. Ledoux, Aspect-oriented programming using reflection, Tech. Rep. 2002-10-3, Ecole des Mines de Douai (Oct. 2002).

[17] N. Bouraqadi, T. Ledoux, Aspect-Oriented Software Development, Addison-Wesley, 2003, Ch. 11 – Supporting AOP using Reflection, (to appear).

[18] G. Kiczales, J. des Rivires, D. G. Bobrow, The Art of the Metaobject Protocol, MIT Press, 1991.

[19] G. Bracha, D. Griswold, Extending smalltalk with mixins, OOPSLA'96 workshop on Extending Smalltalk (October 1996).

[20] A. Goldberg, D. Robson, Smalltalk 80, Vol. The Language and its implementation, Addison-Wesley, 1983.

[21] N. Schärli, S. Ducasse, O. Nierstrasz, A. Black, Traits: Composable units of behavior, in: Proceedings ECOOP 2003, LNCS, Springer Verlag, 2003.

[22] S. Ducasse, N. Schrli, R. Wuyts, Safe and explicit composition of class properties, Tech. Rep. iam-03-007, Institute of Computer Science and Applied Mathematics, University of Berne (Sep. 2003).

Biography

Dr. Noury Bouraqadi joined on 2001 the Ecole des Mines of Douai (France), where he founded the Computer Science Lab (CSL). Since 1995, he has been involved in different industrial and academic projects, using different programming languages including Smalltalk and Java. His research aims at easing development of complex software. For this purpose, Dr. Bouraqadi has been working on reflection, aspect-oriented programming and software components in the context of distributed systems.

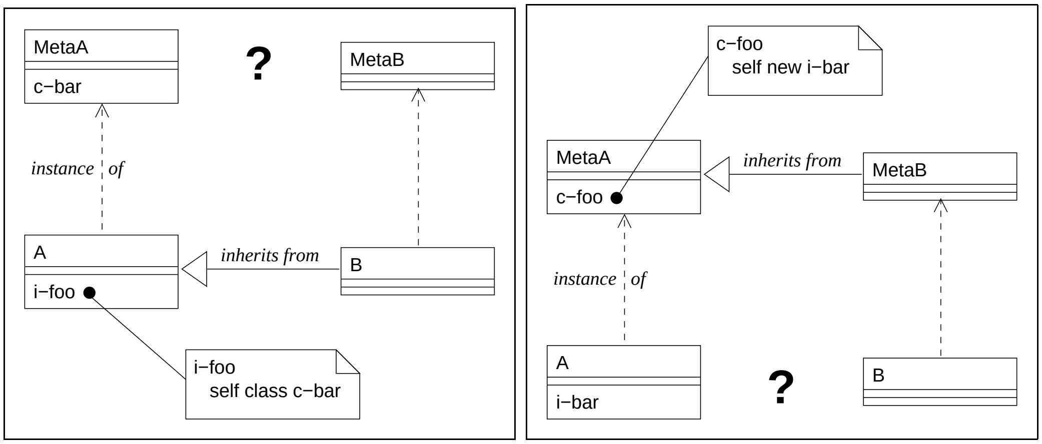

Fig. 1. Examples where class-metaclass compatibility is required

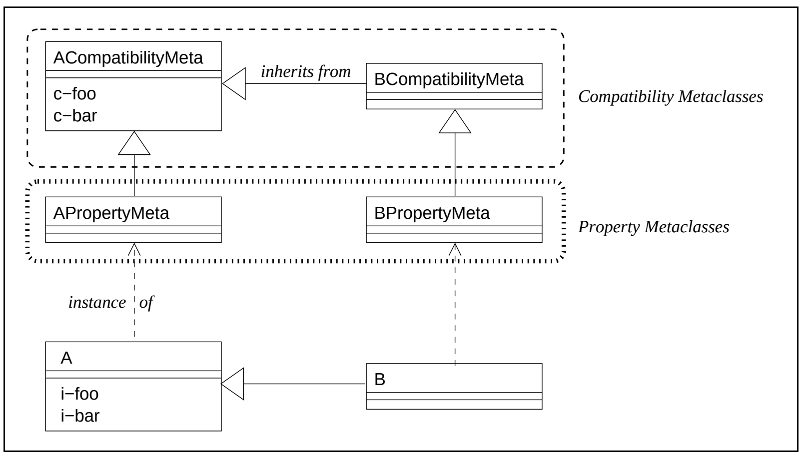

Fig. 2. The compatibility model

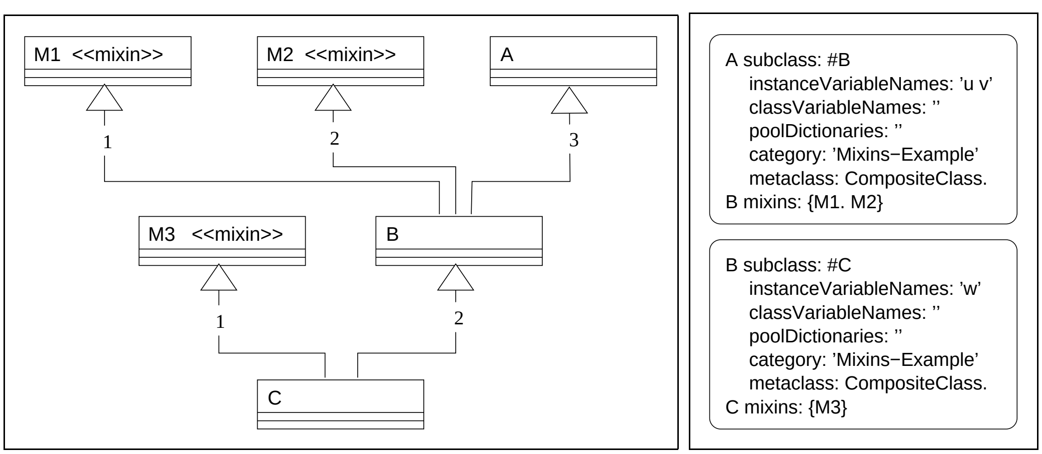

Fig. 3. An inheritance hierarchy with mixins

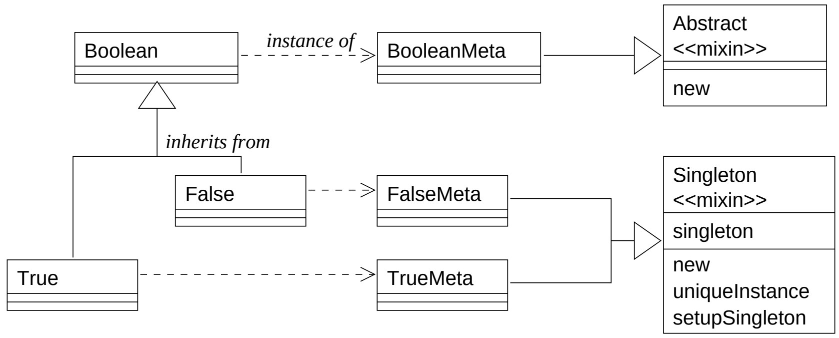

Fig. 4. Enforcing class properties of the Boolean hierarchy using mixins

```
Mixin named: #Abstract
  instanceVariableNames: ''
  category: 'MetaclassTalk-Mixin Library'.

new
  self error: 'Abstract class should not be instantiated'
```

Fig. 5. Implementation of the abstract class property using a mixin

```
Mixin named: #Singleton
  instanceVariableNames: 'singleton'
  category: 'MetaclassTalk-Mixin Library'.

new
  self error: 'Please retrieve my sole instance using the uniqueInstance message.'

uniqueInstance
  singleton ifNil: [self setupSingleton].
  ↑singleton

setupSingleton
  self instanceCount = 0
    ifTrue: [singleton := super new]
    ifFalse: [singleton := self someInstance]
```

Fig. 6. Implementation of the singleton class property using a mixin

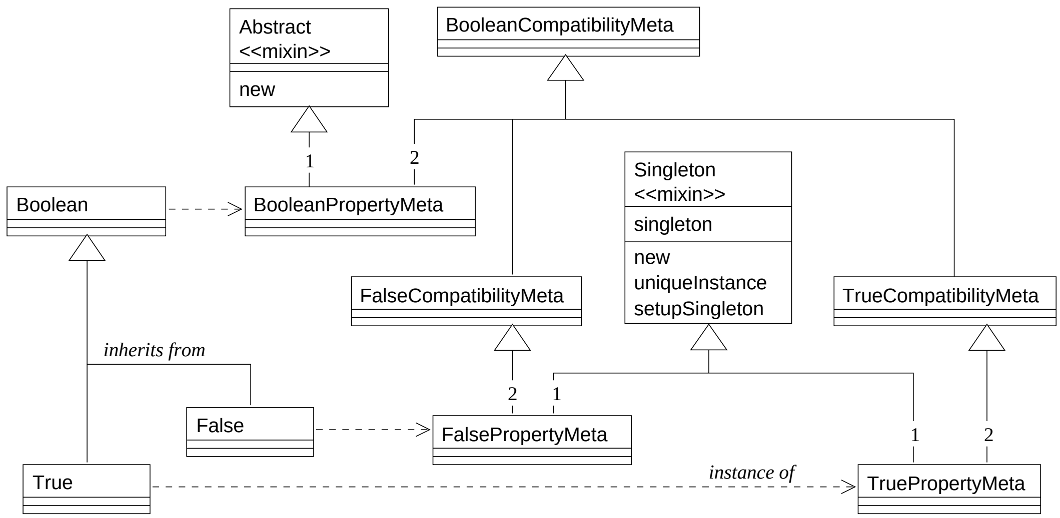

Fig. 7. Ensuring compatibility for the Boolean hierarchy

Classes	**Metaclasses Linearisation Lists**
Boolean	{BooleanPropertyMeta. Abstract. BooleanCompatibilityMeta}
True	{TruePropertyMeta. Singleton. TrueCompatibilityMeta. BooleanCompatibilityMeta}
False	{FalsePropertyMeta. Singleton. FalseCompatibilityMeta. BooleanCompatibilityMeta}

Fig. 8. Linearization lists for metaclasses of the Boolean hierarchy

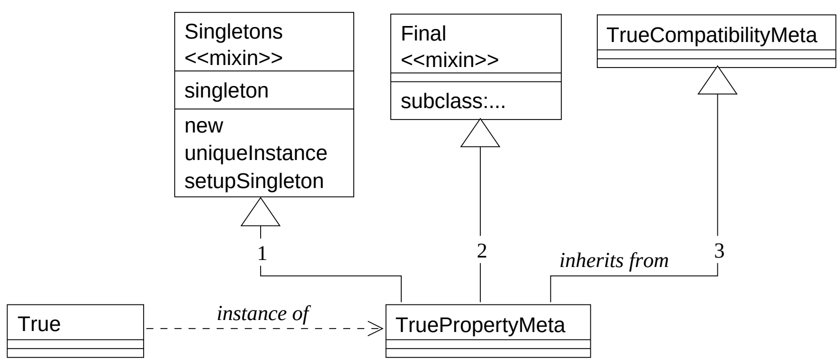

Fig. 9. Class properties composition using mixin-based inheritance

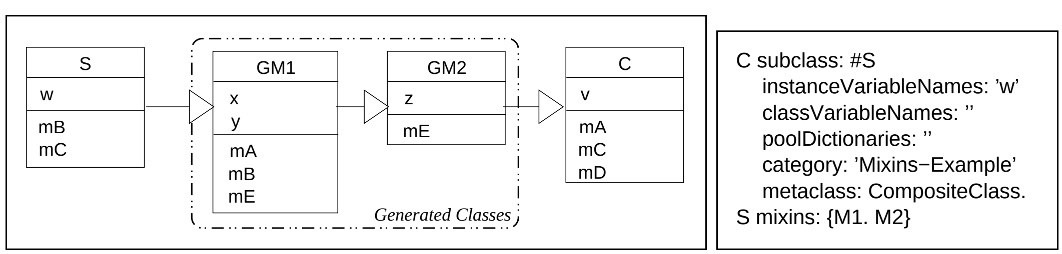

Fig. 10. The actual inheritance hierarchy of a class S inheriting from two mixins M1 and M2