Safe Metaclass Programming

Noury M. N. Bouraqadi-Saâdani
Noury.Bouraqadi@emn.fr
École des Mines de Nantes
BP 20722
44307 Nantes - FRANCE

Thomas Ledoux*
Thomas.Ledoux@emn.fr
École des Mines de Nantes
BP 20722
44307 Nantes - FRANCE

Fred Rivard*
Fred_Rivard@oti.com
École des Mines & OTI Inc. Nantes
BP 20722
44307 Nantes - FRANCE

Abstract

In a system where classes are treated as first class objects, classes are defined as instances of other classes called *metaclasses*. An important benefit of using metaclasses is the ability to assign *properties* to classes (e.g. being abstract, being final, tracing particular messages, supporting multiple inheritance), independently from the base-level code. However, when both inheritance and instantiation are explicitly and simultaneously involved, communication between classes and their instances raises the *metaclass compatibility* issue. Some languages (such as SMALLTALK) address this issue but do not easily allow the assignment of specific properties to classes. In contrast, other languages (such as CLOS) allow the assignment of specific properties to classes but do not tackle the compatibility issue well.

In this paper, we describe a new model of metalevel organization, called *the compatibility model*, which overcomes this difficulty. It allows *safe metaclass programming* since it makes it possible to assign specific properties to classes while ensuring metaclass compatibility. Therefore, we can take advantage of the expressive power of metaclasses to build reliable software. We extend this compatibility model in order to enable safe reuse and composition of class specific properties. This extension is implemented in NEOCLASSTALK, a fully reflective SMALLTALK.

Keywords: Metaclasses, compatibility, class specific properties, class property propagation.

1 Introduction

It has been shown that programming with metaclasses is of great benefit [KAJ+93][Zim96] [BGL98]. An interesting use of metaclasses is the assignment of *specific properties* to classes. For example, a class can be abstract, have a unique instance, trace messages received by its instances, define pre-post conditions on its methods, forbid redefinition of some particular methods... These properties can be implemented using metaclasses, allowing thereby the customization of the classes behavior [LC96].

From an architectural point of view, using metaclasses organizes applications into abstraction levels. Each level describes and controls the level immediately below to which it is causally connected [Mae87]. Reified classes communicate with other objects including their own instances. Thus, classes can send messages to their instances and instances can send messages to their classes. Such message sending is named *inter-level communication* [MMC95].

However, careless inheritance at one level may break inter-level communication resulting in an issue called *the compatibility issue* [BSLR96]. We have identified two symmetrical kinds of compatibility issues. The first one is the *upward compatibility* issue, which was named *metaclass compatibility* by Nicolas Graube [Gra89], and the second one is the *downward compatibility* issue. Both kinds of compatibility issues are important impediments to metaclass programming that one should always be aware of.

*Funded by IBM Global Services - FRANCE
*Since the 1st July 1998: Object Technology International Inc. 2670 Queensview Drive, Ottawa, Ontario, CANADA K2B 8K1.

Currently, none of the existing languages dealing with metaclasses allow the assignment of specific properties to classes while ensuring compatibility. CLOS [KdRB91] allows one to assign any property to classes, but it does not ensure compatibility. On the other hand, both SOM [SOM93] and SMALLTALK [GR83] address the compatibility issue but they introduce a *class property propagation* problem. Indeed, a property assigned to a class is automatically propagated to its subclasses. Therefore, in SOM and SMALLTALK, a class cannot have a specific property. For example, when assigning the abstractness property to a given SMALLTALK class, subclasses become abstract too [BC89]. It follows that users face a dilemma: using a language that allows the assignment of specific class properties without ensuring compatibility, or using a language that ensures compatibility but suffers from the class property propagation problem.

In this paper, we present a model — *the compatibility model* — which allows safe metaclass programming, i.e. it makes it possible to assign specific properties to classes without compromising compatibility. In addition to ensuring compatibility, the compatibility model avoids class property propagation: a class can be assigned specific properties without any side-effect on its subclasses.

We implemented the compatibility model in NEOCLASSTALK, a SMALLTALK extension which introduces many features including explicit metaclasses [Riv96]. Our experiments [Led98][Riv97] showed that the compatibility model allows programmers to fully take advantage of the expressive power of metaclasses. This effort has resulted (i) in a tool that permits a programmer unfamiliar with metaclasses to transparently deal with class specific properties, and (ii) in an approach allowing reuse and composition of class properties.

This paper is organized as follows. Section 2 presents the compatibility issue. We give some examples to show its significance. Section 3 shows how existing programming languages address the compatibility issue, and how they deal with the property propagation problem. Section 4 describes our solution and illustrates it with an example. In section 5, we deal with reuse and composition of class specific properties within the compatibility model. Then, we sketch out the use of the compatibility model for both base-level and meta-level programmers. The last section contains a concluding summary.

2 Inter-level communication and compatibility

We define inter-level communication as any message sending between classes and their instances (see Figure 1). Indeed, class objects can interact with other objects by sending and receiving messages. In particular, an instance can send a message to its class and a class can send a message to some of its instances. We use SMALLTALK as an example to illustrate this issue[1].

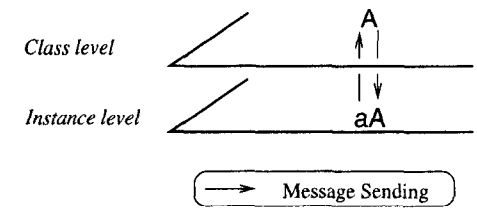

Figure 1: Inter-level communication

Two methods allow inter-level communication in SMALLTALK: **new** and **class**. When one of them is used, the involved objects belong to different levels of abstraction[2]:

- An object receiving the **class** message returns its class. Then, the **class** method makes it possible to go one level up. The following two instance methods — excerpted from Visual Works SMALLTALK — include message sending to the receiver's class.

```
⋆ message name is sent to the class:
Object≫printOn: aStream
  | title |
  title := self class name.
  ...
⋆ message daysInYear: is sent to the class:
Date≫daysInYear
  "Answer the number of days in the year
  represented by the receiver."
  ↑ self class daysInYear: self year
```

[1] We use the SMALLTALK syntax and terminology throughout this paper.

[2] Static measures we made over a Visual Works SMALLTALK image show that inter-level communication is very frequent. 25% of classes include instance methods referencing the class and 24% of metaclasses define methods referencing an instance.

- A class receiving the **new** message returns a new instance. Therefore, the **new** method makes it possible to go one level down. The following two class methods include message sending to the newly created instances.

```
⋆ message at:put: is sent to a new instance:
ArrayedCollection class≫with: anObject
  | newCollection |
  newCollection := self new: 1.
  newCollection at: 1 put: anObject.
  ↑newCollection
⋆ message on: is sent to a new instance:
Browser class≫openOn: anOrganizer
  self openOn: (self new on: anOrganizer) with-
TextState: nil
```

Thus, inter-level communication in SMALLTALK is materialized by sending the messages **new** and **class**. Other languages where classes are reified (such as CLOS and SOM) also allow similar message sending.

Since these inter-level communication messages are embedded in methods, they are inherited whenever methods are inherited. Ensuring *compatibility* means making sure that these methods will not induce any failure in subclasses, i.e. all sent messages will always be understood. We have identified two kinds of compatibility: *upward compatibility*[3] and *downward compatibility.*

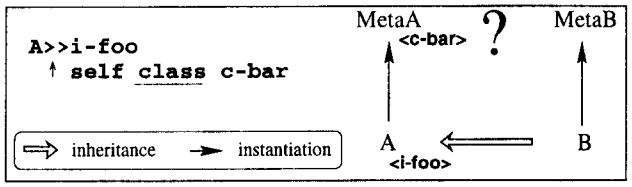

Figure 2: Compatibility need to be ensured at a higher level

2.1 Upward compatibility

Suppose A implements a method i-foo that sends the c-bar message to the class of the receiver (see Figure 2). B is a subclass of A. When i-foo is sent to an instance of B, the B class receives the c-bar message. In order to avoid any failure, B should understand the c-bar message (i.e. MetaB should implement or inherit a method c-bar).

[3]Nicolas Graube named this issue *metaclass compatibility*[Gra89].

Definition of upward compatibility:
Let B be a subclass of the class A, MetaB the metaclass of B, and MetaA the metaclass of A.

Upward compatibility is ensured for MetaB and MetaA iff: every possible message that does not lead to an error for any instance of A, will not lead to an error for any instance of B.

2.2 Downward compatibility

Suppose MetaA implements a method c-foo that sends the i-bar message to a newly created instance (see Figure 3). MetaB is created as a subclass of MetaA. When c-foo is sent to B (an instance of MetaB), B will create an instance which will receive the i-bar message. In order to avoid any failure, instances of B should understand the i-bar message (i.e. B should implement or inherit the i-bar method).

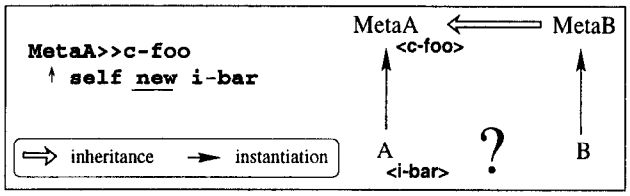

Figure 3: Compatibility need to be ensured at a lower level

Definition of downward compatibility:
Let MetaB be a subclass of the metaclass MetaA.

Downward compatibility is ensured for two classes B instance of MetaB and A instance of MetaA iff: every possible message that does not lead to an error for A, will not lead to an error for B.

3 Existing models

We will now show why none of the known models allow the assignment of specific properties to classes while ensuring compatibility.

3.1 CLOS

When (re)defining a class in CLOS, the **validate-superclass** generic function is called, before the direct superclasses are stored [KdRB91]. As a default, **validate-superclass** returns true if the meta-

class of the new class is the same as the metaclass of the superclass[4], i.e. classes and their subclasses must have the same metaclass. Therefore, incompatibilities are avoided but metaclass programming is very constrained.

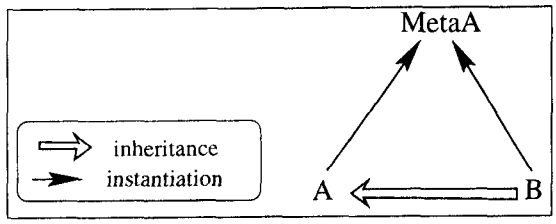

Figure 4: By default in CLOS, subclasses must share the same metaclass as their superclass

Figure 4 shows a hierarchy of two classes that illustrates the CLOS default compatibility management policy. Since class B inherits from A, B and A must have the same metaclass.

In order to allow the definition of classes with different behaviors, programmers usually redefine the validate-superclass method to make it always return true. Thus, CLOS programmers can have total freedom to use a specific metaclass for each class. So, they can assign specific properties to classes, but the trade-off is that they need to be always aware of the compatibility issue.

3.2 SOM

SOM is an IBM CORBA compliant product which is based on a metaclass architecture [DF94b]. The SOM kernel follows the OBJVLISP model [Coi87]. SOM metaclasses are explicit and can have many instances. Therefore, users have complete freedom to organize their metaclass hierarchies.

3.2.1 Compatibility issue in SOM

SOM encourages the definition and the use of explicit metaclasses by introducing a unique concept named *derived metaclasses* which deals with the *upward compatibility* issue [DF94a] At compile-time, SOM automatically determines an appropriate metaclass that ensures upward compatibility. If needed, SOM automatically creates a new metaclass named a *derived metaclass* to ensure upward compatibility.

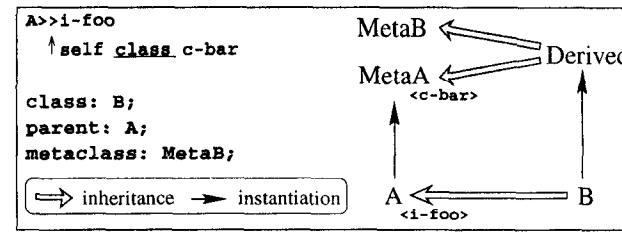

Figure 5: SOM ensures upward compatibility using derived metaclasses

Suppose that we want to create a class B, instance of MetaB and subclass of A. SOM will detect an upward compatibility problem, since MetaB does not inherit from the metaclass of A (MetaA). Therefore, SOM automatically creates a derived metaclass (Derived), using multiple inheritance in order to support all necessary class methods and variables[5]. Figure 5 shows the resulting construction. When an instance of B receives i-foo, it goes one level higher and sends c-bar to the B class. B understands the c-bar message since its metaclass (i.e. Derived) is a derived metaclass which inherits from both MetaB and MetaA.

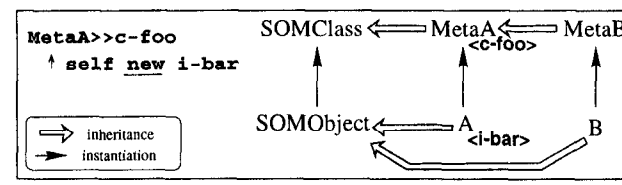

Figure 6: SOM downward compatibility failure example

SOM does not provide any policy or mechanism to handle *downward compatibility*. Suppose that MetaB is created as a subclass of MetaA (see Figure 6). The c-foo method which is inherited by MetaB sends the i-bar message to a new instance. When the B class receives the c-foo message, a runtime error will occur because its instances do not understand the i-bar message.

[4]In fact it also returns true if the superclass is the class named t, or if the metaclass of one argument is **standard-class** and the metaclass of the other is **funcallable-standard-class**.

[5]The semantics of derived metaclasses guarantees that the declared metaclass takes precedence in the resolution of multiple inheritance ambiguities (i.e. **MetaB** before **MetaA**). Besides, it ensures the instance variables of the class are correctly initialized by the use of a complex mechanism.

3.2.2 Class property propagation in SOM

SOM does not allow the assignment of a property to a given class, without making its subclasses be assigned the same property. We name this defect *the class property propagation* problem. In the following example, we illustrate how derived metaclasses implicitly cause undesirable propagation of class properties.

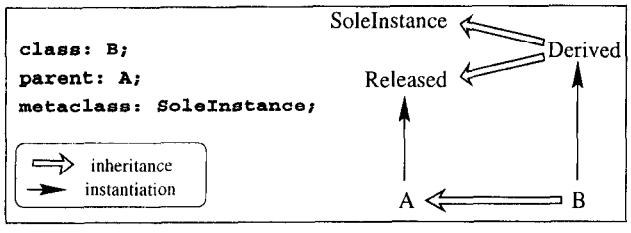

Figure 7: Class property propagation in SOM

Suppose that the A class of Figure 7 is a released class, i.e. it should not be modified any more. This property is useful in multi-programmer development environments for versionning purposes. In order to avoid any change, A is an instance of the Released metaclass. Let B a class that has a unique instance: B is an instance of the SoleInstance metaclass. But as B is a subclass of A, SOM creates B as instance of an automatically created derived metaclass which inherits from both SoleInstance and Released. Thus, as soon as B is created, it is automatically "locked" and acts like a released class. So, we cannot define any new method on it!

3.3 Smalltalk-80

In SMALLTALK, metaclasses are partially hidden and automatically created by the system. Each metaclass is non-sharable and strongly coupled with its sole instance. So, the metaclass hierarchy is parallel to the class hierarchy and is implicitly generated when classes are created.

3.3.1 Compatibility issue in Smalltalk-80

Using parallel inheritance hierarchies, the SMALLTALK model ensures both upward and downward compatibility. Indeed, any code dealing with **new** or **class** methods, is inherited and works properly.

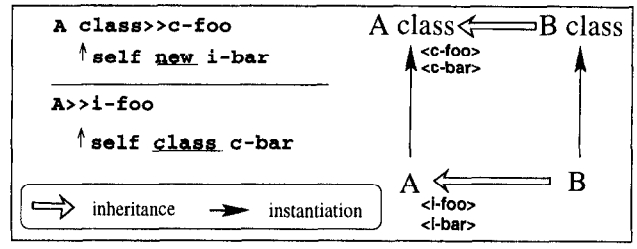

Figure 8: SMALLTALK ensures both upward and downward compatibilities

When one creates the B class, a subclass of A (see Figure 8), SMALLTALK automatically generates the metaclass of B ("B class"[6]), as a subclass of "A class", the metaclass of A. Suppose A implements a method **i-foo** that sends **c-bar** to the class of the receiver. If **i-foo** is sent to an instance of B, the B class receives the **c-bar** message. Thanks to the parallel hierarchies, the B class understands the **c-bar** message, and upward compatibility is ensured. In a similar manner, downward compatibility is ensured thanks to the parallel hierarchy.

3.3.2 Class property propagation in Smalltalk-80

Since metaclasses are automatically and implicitly managed by the system, SMALLTALK drastically reduces the opportunity to change class behaviors, making metaclass programming "anecdotal". As with SOM, SMALLTALK does not allow the assignment of a property to a class without propagating it to its subclasses.

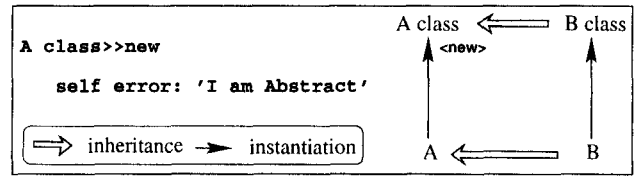

Figure 9: Class property propagation in SMALLTALK

In Figure 9, the A class is abstract since its subclasses must implement some methods to complete the instance behavior. B is a concrete class as it implements the whole set of these methods. Suppose

[6]The name of a SMALLTALK metaclass is the name of its unique instance postfixed by the word 'class'.

that we want to enforce the property of abstractness of A. In order to forbid instantiating A, we define the class method A class≫new which raises an error. Unfortunately, "B class" inherits the method new from "A class". As a result, attempting to create an instance of B raises an error[7]!

4 The compatibility model

Among the previous models, only the SMALLTALK one with its parallel hierarchies ensures full compatibility. However, it does not allow the assignment of specific properties to classes. On the other hand, only the CLOS model allows the assignment of specific properties to classes. Unfortunately, it does not ensure compatibility. We believe that these two goals can both be achieved by a new model which makes a clear separation between compatibility and class specific properties.

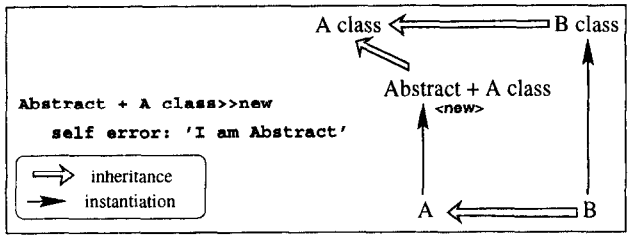

Figure 10: Avoiding the propagation of abstractness

We illustrate this idea of separation of concerns by refactoring the example of Figure 9. We create a new metaclass named "Abstract + A class" as a subclass of "A class" (see Figure 10). The A class is redefined as an instance of this new metaclass. As "Abstract + A class" redefines the new method to raise an error, A cannot have any instance. However, since "B class" is not a subclass of "Abstract + A class", the B class remains concrete. The generalization of this scheme is our solution, named *the compatibility model.*

In the remainder of this paper, names of metaclasses defining some class property are denoted with the concatenation of the property name, the + symbol and the superclass name. For example, "Abstract + A class" is a subclass of "A class" that defines the property of abstractness named Abstract.

4.1 Description of the compatibility model

The compatibility model extends the SMALLTALK model by separating two concerns: compatibility and specific class properties. A metaclass hierarchy parallel to the class hierarchy ensures both upward and downward compatibility like in SMALLTALK. An extra metaclass "layer" is introduced in order to *locally* assign any property to classes. Classes are instances of metaclasses belonging to this layer. So, the compatibility model is based on two "layers" of metaclasses, each one addressing a unique concern:

Compatibility concern: This issue is addressed by the metaclasses organized in a hierarchy parallel to the class hierarchy. We name such metaclasses: *compatibility metaclasses.* They define all the behavior that must be propagated to all (sub)classes. All class methods which send messages to instances should be defined in these metaclasses. Besides, all messages sent to classes by their instances should be defined in these metaclasses too.

Specific class properties concern: This issue is addressed by metaclasses that define the class specific properties. We name such metaclasses: *property metaclasses.* A class with a specific property is instance of a property metaclass which inherits from the corresponding compatibility metaclass. The property metaclass is not joined to other property metaclasses, since it defines a property specific to the class.

Figure 11 shows[8] the compatibility model applied to a hierarchy consisting of two classes: A and B. They are respectively instances of the "AProperty + AClass" and "BProperty + BClass" metaclasses. "AProperty + AClass" defines properties specific to class A, while "BProperty + BClass" defines properties specific to class B. As "AProperty + AClass" and "BProperty + BClass" are not joined

[7] This example is deliberately simple, and one could avoid this problem by redefining new in "B class". But, this solution is a kind of inheritance anomaly [MY93] that increases maintenance costs.

[8] Compatibility metaclasses are surrounded with a dashed line and property metaclasses are drawn inside a grey shape.

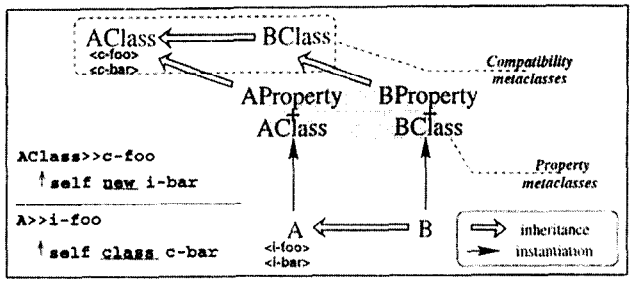

Figure 11: The compatibility model

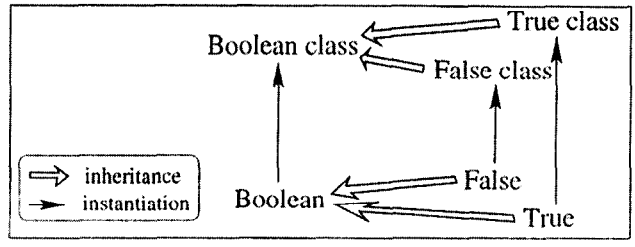

Figure 12: The Boolean hierarchy of SMALLTALK

by any link, class property propagation does not occur. Thus, A and B can have distinct properties.

Since "AProperty + AClass" and "BProperty + BClass" are subclasses of the AClass and BClass metaclasses, both upward and downward compatibility are guaranteed. Suppose that A defines two instance methods i-foo and i-bar. The i-foo method sends the c-bar message to the class of the receiver. The i-bar method is sent to a new instance by the c-foo method. Because the AClass and BClass metaclass hierarchy is parallel to the A and B class hierarchy, inter-level communication failure is avoided.

4.2 Example: Refactoring the Smalltalk-80 Boolean hierarchy

The Boolean hierarchy of SMALLTALK[9] is depicted in Figure 12. Boolean is an abstract class which defines a protocol shared by True and False. True and False are concrete classes that cannot have more than one instance. These properties (i.e. abstractness and having a sole instance) are implicit in SMALLTALK. Using the compatibility model, we refactor the Boolean hierarchy to emphasize them.

We first create "Boolean class", which is a compatibility metaclass. The second step consists of creating the property metaclass "Abstract + Boolean class", which enforces the Boolean class to be abstract. Finally, we build the Boolean class by instantiating the "Abstract + Boolean class" metaclass.

To refactor the False class, we first create the "False class" metaclass, as a subclass of "Boolean class" to ensure the compatibility. The second step consists of creating the property metaclass "SoleInstance + False class", which enforces the False class to have at most one instance. At last, we create the False class by instantiating the "SoleInstance + False class" metaclass. The True class is refactored in the same way. The result of rebuilding the whole hierarchy of Boolean is shown in Figure 13.

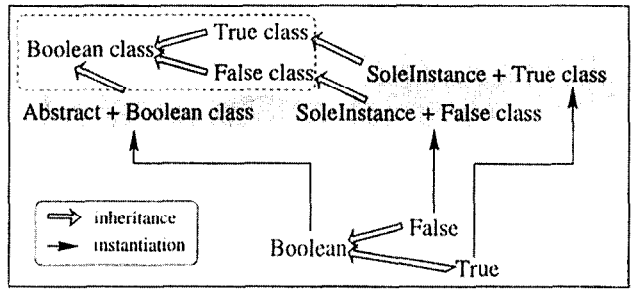

Figure 13: The Boolean hierarchy after refactoring

5 Reuse and composition within the compatibility model

We have experimented the compatibility model in NEOCLASSTALK[10] [Riv97], a fully reflective SMALLTALK. We quickly faced the need of *class property reuse and composition*. Indeed, unrelated classes belonging to different hierarchies can have the same properties, and a given class can have many properties.

In the previous section, both the True class and the False class have the same property: having a unique instance. Besides, we assigned only one property to each class of the Boolean hierarchy.

[9] We prefer this academic example to emphasize our ideas rather than a more complex example which should require a more detailed presentation.

[10] NEOCLASSTALK and all related papers can be downloaded from http://www.emn.fr/cs/object/tools/neoclasstalk/neoclasstalk.html

But, a class need to be assigned many properties. For example, the **False** class must not only have a unique instance, but it also should not be subclassed (such a class is **final** in JAVA terminology). So, we have to reuse and compose these class properties with respect to our compatibility model.

In this section, we propose an extension of our compatibility model that deals with reuse and composition of class properties. Any language where classes are treated as regular objects may integrate our extended compatibility model. NEOCLASSTALK has been used as a first experimentation platform.

5.1 Reuse of class properties

In SMALLTALK, since metaclasses behave in a different way than classes, they are defined as instances of a particular class, a *meta-metaclass*, called **Metaclass**. **Metaclass** defines the behavior of all metaclasses in SMALLTALK. For example, the name of a metaclass is the name of its sole instance postfixed by the word 'class'.

```
Metaclass≫name
  ↑thisClass name, ' class'
```

We take advantage of this concept of meta-metaclasses to reuse class properties. Since metaclasses implementing different properties have different behaviors, we need one meta-metaclass for each class property. Property metaclasses defining the same class property are instances of the same meta-metaclass.

When a property metaclass is created, the meta-metaclass initializes it with the definition of the corresponding class property. Thus, the code (instance variables, methods, ...) corresponding to the definition of the class property, is automatically generated. Reuse is achieved by creating property metaclasses defining the same class property as instances of the same meta-metaclass, i.e. they are initialized with the same class property definition (an example of such an initialization is given in section 5.4.2).

The root of the meta-metaclass hierarchy named **PropertyMetaclass** describes the default structure and behavior of property metaclasses. For example, the name of a property metaclass is built from the property name and the superclass name:

```
PropertyMetaclass≫name
  ↑self class name, '+', self superclass name
```

In the refactored **Boolean** hierarchy of section 4.2, both "**SoleInstance + False class**" and "**SoleInstance + True class**" define the property of having a unique instance. Reuse is achieved by defining both "**SoleInstance + False class**" and "**SoleInstance + True class**" as instances of **SoleInstance**, a subclass of **PropertyMetaclass** (see Figure 14).

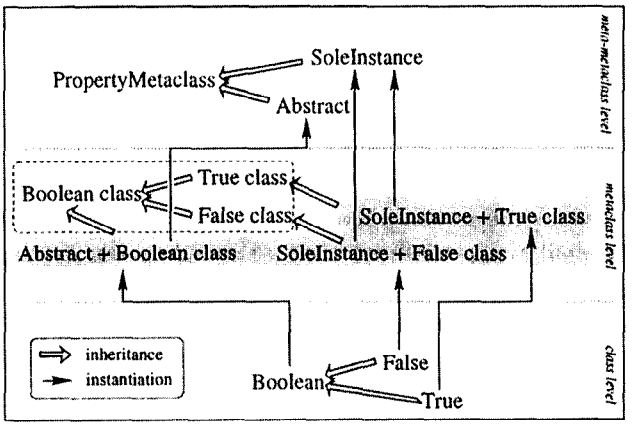

Figure 14: Reuse properties in the **Boolean** hierarchy

5.2 Composition of class properties

Since a given class can have many properties, the model must support the composition of class properties. We chose to use many property metaclasses organized in a single inheritance hierarchy, where each metaclass implements one specific class property.

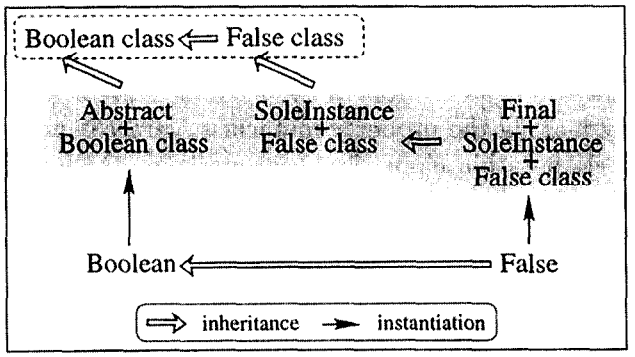

Figure 15: Assigning two properties to False

To illustrate this idea, we modify the instantiation link for the False class (see Figure 15). We define two property metaclasses, one for each property. The first property metaclass is "SoleInstance + False class", which inherits from the compatibility metaclass "False class". The second one is "Final + SoleInstance + False class", which is the class of False. It is defined as a subclass of "SoleInstance + False class". The resulting scheme respects the compatibility model: it allows the assignment of two specific properties to the False class and still ensures compatibility.

5.2.1 Conflict management

The solution of the property metaclasses composition issue is not trivial. Indeed, it is necessary to deal with conflicts that arise when composing different property metaclasses. When using inheritance to compose property metaclasses, two kinds of conflicts can arise: *name conflicts* and *value conflicts* [DHH+95].

Name conflicts happen when orthogonal property metaclasses define instance variables or methods which have the same name. Two property metaclasses are orthogonal when they define unrelated class properties. Name conflicts for both instance variables and methods are avoided by adapting the definition of a new property metaclass according to its superclasses. For example, although the two property metaclasses "SoleInstance + False class" and "SoleInstance + True class" define the same property for their respective instances (classes False and True), they may use different instance variable names or method names.

Value conflicts happen when non-orthogonal property metaclasses define methods which have the same name. Most of these conflicts are avoided by making the property metaclass hierarchy act as a cooperation chain, i.e. a property metaclass explicitly refer to the overridden methods defined in its superclasses[11]. Therefore, each property metaclass acts like a mixin [BC90].

[11] In NeoClasstalk, as in Smalltalk, this is achieved using the pseudo-variable super.

5.2.2 Example of cooperation between property metaclasses

Suppose that we want to assign two specific properties to the False class of Figure 16: (i) *tracing* all messages (Trace) and (ii) having *breakpoints* on particular methods (BreakPoint). These two properties deal with the message handling which is based in NeoClasstalk on the technique of the "method wrappers" described in [Duc98] and [BFJR98]. The executeMethod:receiver:arguments: method is the entry point to handle messages in NeoClasstalk, i.e. customizing executeMethod:receiver:arguments: allows a specialization of the message sending[12]. Thus, when the object false receives a message, the class False receives the message executeMethod:receiver:arguments:.

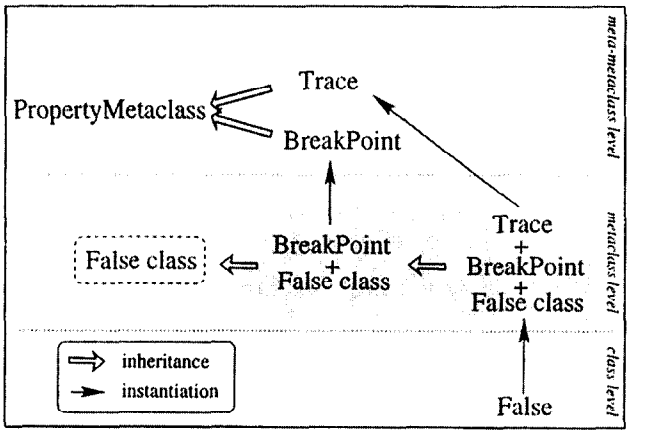

Figure 16: Composition of non-orthogonal properties

According to the inheritance hierarchy, (1) the trace is first done, then (2), by the use of super, the breakpoint is performed, and (3) a regular method application is finally executed (again called using super).

- (3) StandardClass≫executeMethod: method receiver: rec arguments: args
  ...
- (2) BreakPoint+False class≫executeMethod: method receiver: rec arguments: args
  method selector == stopSelector
  ifTrue: [self halt: 'Breakpoint for ', stopSelector].
  ↑super executeMethod: method receiver: rec arguments: args

[12] A default executeMethod:receiver:arguments: method is provided by StandardClass (the root of all metaclasses in NeoClasstalk) which just applies the method on the receiver with the arguments.

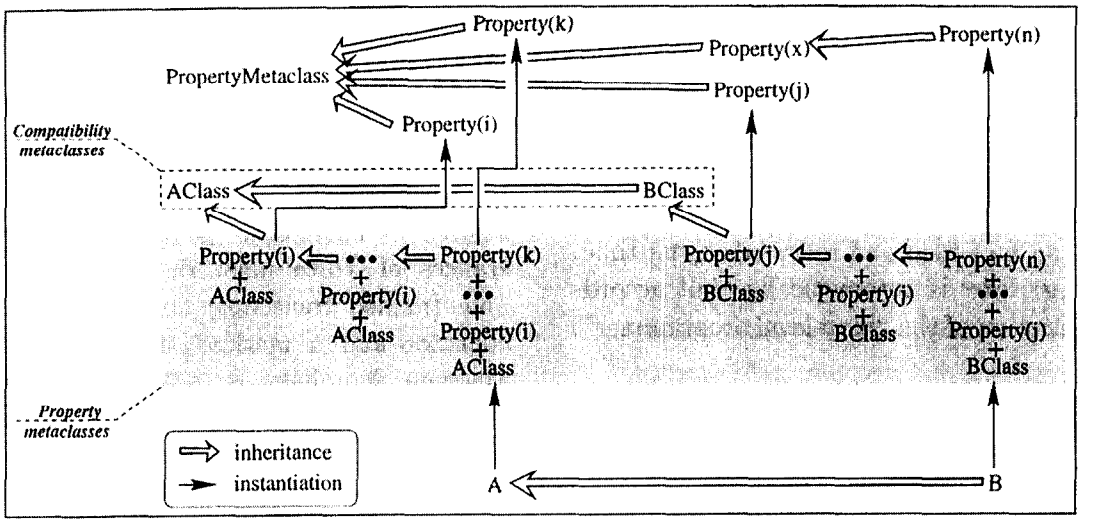

Figure 17: The Extended Compatibility Model

- (1) Trace+BreakPoint+False class≫executeMethod: method receiver: rec arguments: args

  self transcript show: method selector; cr.
  ↑super executeMethod: method receiver: rec arguments: args

5.3 The extended compatibility model

Generalizing previous examples allows us to define the *extended compatibility model* (see Figure 17) which enables reusing and composing class properties. Each property metaclass defines the instance variables and methods involved in a unique property. Property metaclasses specific to a given class are organized in a single hierarchy. The root of this hierarchy is a subclass of a compatibility metaclass[13]. Each property metaclass is an instance of a meta-metaclass which describes a specific class property, allowing its reuse.

Metaclass creation, composition and deletion are managed automatically with respect to the extended compatibility model. Each time a new class is created, a new compatibility metaclass is automatically created. This can be done in the same way that SMALLTALK builds its parallel metaclass hierarchy. The assignment of a property to this class results in the insertion of a new metaclass into its property metaclass hierarchy. This insertion is made in two steps[14]:

1. first, the new property metaclass becomes a subclass of the last metaclass of the property metaclass hierarchy;
2. then, the class becomes instance of this new property metaclass.

NEOCLASSTALK provides protocols for dynamically changing the class of an object (**changeClass:**) and the superclass of a class (**superclass:**) [Riv97]. Thus, the implementation of these two steps is immediate in NEOCLASSTALK, and is provided by the **composeWithPropertiesOf:** method.

```
PropertyMetaclass≫composeWithPropertiesOf: aClass
    self superclass: aClass class.
    aClass changeClass: self.
```

5.4 Programming within the extended compatibility model

We distinguish two kinds of programmers: (i) "base level programmers" who implement applications using the language and development tools, and (ii) "meta level programmers" for whom the language itself is the application.

[13] This single hierarchy may be compared to an explicit linearization of property metaclasses composed using multiple inheritance [DHHM94].

[14] The removal of a property metaclass is done in a symmetrical way.

5.4.1 Base Level Programming

To make our model easy to use for a "base-level programmer", the NEOCLASSTALK programming environment includes a tool that allows one to assign different properties to a given class using a SMALLTALK-like browser (see Figure 18). These properties can be added and removed at run-time. The metaclass level is automatically built according to the selection of the "base-level programmer".

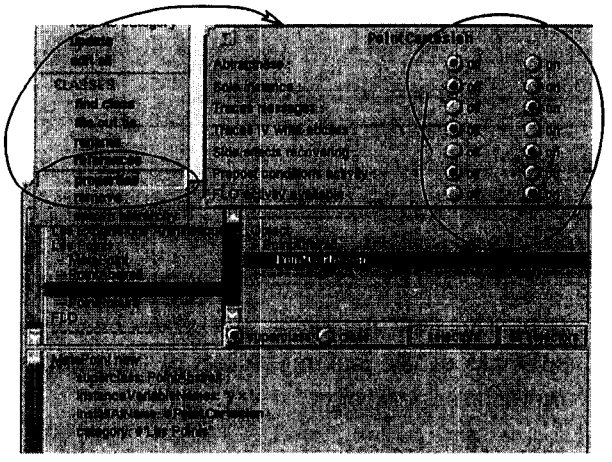

Figure 18: Specific properties assigned to a class using a browser

5.4.2 Meta Level Programming

In order to introduce new class properties, "meta-level programmers" must create a subclass of the **PropertyMetaclass** meta-metaclass. This new meta-metaclass stores the instance variables and the methods that should be defined by its instances (property metaclasses). When this new meta-metaclass is instantiated, the previous instance variables are added to the resulting property metaclass and the methods are compiled[15] at initialization time[16].

For example, the evaluation of the following expression creates a property metaclass — instance of the meta-metaclass **Trace** — that assigns the trace property to the **True** class.

```
Trace new composeWithPropertiesOf: True
```

In order to achieve the trace, messages must be captured and then logged in a text collector. Therefore, property metaclasses instances of **Trace** must define an instance variable (named **transcript**) corresponding to a text collector and a method that handles messages. Message handling is achieved using the **executeMethod:receiver:arguments:** method which source code was already presented in 5.2.2. These definitions are generated when the property metaclasses are initialized, i.e. using the **initialize** method of the **Trace** meta-metaclass:

```
Trace≫initialize
  super initialize.
  self instanceVariableNames:' transcript '.
  self generateExecuteMethodReceiverArguments.
```

6 Conclusion

Considering classes as first class objects organizes applications in different abstraction levels, which inevitably raises upward and downward compatibility issues. Existing solutions addressing the compatibility issues (such as SMALLTALK) do not allow the assignment of specific properties to a given class without propagating them to its subclasses.

The *compatibility model* proposed in this paper addresses the compatibility issue and allows the assignment of specific properties to classes without propagating them to subclasses. This is achieved thanks to the separation of the two involved concerns: compatibility and class properties. Upward and downward compatibilities are ensured using the *compatibility metaclass* hierarchy that is parallel to the class hierarchy. The *property metaclasses*, allowing the assignment of specific properties to classes, are subclasses of these compatibility metaclasses. Therefore, we can take advantage of the expressive power of metaclasses to define, reuse and compose class properties in a environment which supports *safe metaclass programming.*

Class properties improve readability, reusability and quality of code by increasing *separation of concerns* [HL95] [Lie96] [KLM+97]. Indeed, they

[15] A faster solution consists of doing the compilation only once, resulting in proto-methods [Riv97]. Thus, when the property metaclass gets initialized, proto-methods are "copied" into the method dictionary of the property metaclass, allowing a fast instantiation of meta-metaclasses.

[16] This assumes that initialization is part of the creation process, which is true in almost every language. This is traditionnally achieved in SMALLTALK by the redefinition of **new** into **super new initialize** [SKT96].

allow a better organization of class libraries and frameworks for designing reliable software. We are strongly convinced that our compatibility model enables separation of concerns based on the metaclass paradigm. Therefore, it promotes building reliable software which is easy to reuse and maintain.

Acknowledgments

The authors are grateful to Mathias Braux, Pierre Cointe, Stéphane Ducasse, Nick Edgar, Philippe Mulet, Jacques Noyé, Nicolas Revault, and Mario Südholt for their valuable comments and suggestions. Special thanks to the anonymous referees who provided detailed and thought-provoking comments.

References

[BC89] Jean-Pierre Briot and Pierre Cointe. Programming with Explicit Metaclasses in Smalltalk. In *Proceedings of OOPSLA'89*, pages 419–431, New Orleans, Louisiana, USA, October 1989. ACM.

[BC90] Gilad Bracha and William Cook. Mixin-based Inheritance. In *Proceedings of ECCOP/OOPSLA'90, Ottawa, Canada*, pages 303–311, October 1990.

[BFJR98] John Brant, Brian Foote, Ralph E. Johnson, and Donald Roberts. Wrappers to the Rescue. In *Proceedings of ECOOP'98*, July 1998.

[BGL98] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Löhr. Concurrency and Distribution in Object Oriented Programming. *ACM Computer Surveys*, 1998. to appear.

[BSLR96] Noury Bouraqadi-Saâdani, Thomas Ledoux, and Fred Rivard. Metaclass Composability. In *ECOOP'96 workshop : "Composability Issues in Object Orientation"*, Linz, Austria, July 1996.

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

[DF94a] Scott Danforth and Ira R. Forman. Derived Metaclasses in SOM. In *Proceedings of TOOLS EUROPE'94*, pages 63–73, Versailles, France, 1994.

[DF94b] Scott Danforth and Ira R. Forman. Reflections on Metaclass Programming in SOM. In *Proceedings of OOPSLA'94*, pages 440–452, October 1994.

[DHH+95] Roland Ducournau, Michel Habib, Marianne Huchard, Marie-Laure Mugnier, and Amedeo Napoli. Le point sur l'héritage multiple. *Techniques et Sciences Informatique*, 14(3):309–345, 1995. (In french).

[DHHM94] Roland Ducournau, Michel Habib, Marianne Huchard, and Marie-Laure Mugnier. Proposal for a Monotonic Multiple Inheritance Linearization. In *Proceedings of OOPSLA'94*, pages 164–175, Portland, Oregon, October 1994.

[Duc98] Stéphane Ducasse. Evaluating Message Passing Control Techniques in Smalltalk. *Journal of Object-Oriented Programming*, 1998. to appear.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80, The language and its implementation.* Addison Wesley, Readings, Massachusetts, 1983.

[Gra89] Nicolas Graube. Metaclass Compatibility. In *Proceedings of OOPSLA'89*, pages 305–315, New Orleans, Lousiana, October 1989.

[HL95] Walter L. Hürsch and Cristina Videira Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, February 1995.

[KAJ+93] Gregor Kiczales, J. Michael Ashley, Luis H. Rodriguez Jr., Amin Vahdat, and Daniel G. Bobrow. *"Object-Oriented Programming: The CLOS Perspective" edited by Andreas Pæpcke*, chapter Metaobject Protocols: Why We Want Them and What Else They Can Do, pages 101–118. The MIT Press, Cambridge, Massachusetts, 1993.

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

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97*, number 1241 in LNCS, pages 220–242. Springer-Verlag, June 1997.

[LC96] Thomas Ledoux and Pierre Cointe. Explicit Metaclasses As a Tool for Improving the Design of Class Libraries. In *Proceedings of ISOTAS'96, LNCS 1049*, pages 38–55, Kanazawa, Japan, March 1996. Springer-Verlag.

[Led98] Thomas Ledoux. *Reflection and Distributed Systems : an Experiment with CORBA and Smalltalk.* PhD thesis, Université de Nantes, March 1998. (In french. Réflexion dans les systèmes répartis : application à CORBA et Smalltalk).

[Lie96] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

[Mae87] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87*, pages 147–155, Orlando, Florida, 1987. ACM.

[MMC95] Philippe Mulet, Jacques Malenfant, and Pierre Cointe. Towards a Methodology for Explicit Composition of MetaObjects. In *Proceedings of OOPSLA'95*, pages 316–330, Austin, Texas, October 1995.

[MY93] Satoshi Matsuoka and Akinori Yonezawa. *Research Directions in Concurrent Object-Oriented Programming*, chapter Analysis of inheritance anomaly in object-oriented concurrent programming languages. MIT Press, 1993.

[Riv96] Fred Rivard. A New Smalltalk Kernel Allowing Both Explicit and Implicit Metalclass Programming. OOPSLA'96, Workshop : Extending the Smalltalk Language, October 1996.

[Riv97] Fred Rivard. *Object Behavioral Evolution Within Class Based Reflective Languages.* PhD thesis, Université de Nantes, June 1997. (In french. Évolution du Comportement des Objets dans les Langages à Classes Réflexifs).

[SKT96] Suzanne Skublicks, Edward J. Klimas, and David A. Thomas. *Smalltalk with Style.* Prentice Hall, 1996.

[SOM93] IBM. *SOMobjects Developer Toolkit Users Guide release 2.0*, second edition, June 1993.

[Zim96] Chris Zimmermann, editor. *Advances in Object-Oriented Metalevel Architectures and Reflection.* CRC Press, 1996.