Uniform and Safe Metaclass Composition [⋆]

Stéphane Ducasse [a] Nathanael Schärli [a] Roel Wuyts [b]

[a] *Software Composition Group, IAM-Universität Bern, Switzerland*

[b] *Decomp Laboratory, Université Libre de Bruxelles, Belgium*

Abstract

In pure object-oriented languages, classes are objects, instances of other classes called *metaclasses*. In the same way as classes define the properties of their instances, metaclasses define the properties of classes. It is therefore very natural to wish to reuse class properties, utilizing them amongst several classes. However this introduced *metaclass composition problems*, *i.e.*, code fragments applied to one class may break when used on another class due to the inheritance relationship between their respective metaclasses.

Numerous approaches have tried to solve metaclass composition problems, but they always resort to an *ad-hoc* manner of handling conflicting properties, alienating the meta-programmer. We propose a *uniform* approach that represents class properties as *traits*, groups of methods that act as a unit of reuse from which classes are composed. Like all the other classes in the system, metaclasses are composed out of traits. This solution supports the reuse of class properties, and their *safe* and *automatic* composition based on *explicit* conflict resolution. The paper discusses traits and our solution, shows concrete examples implemented in the Smalltalk environment *Squeak*, and compares our approach with existing models for composing class properties.

Key words: Metaclass composition, traits, reflective kernel, reuse, mixins

1 Reusing class properties

In class-based object-oriented programming, classes are used as instance generators and to implement the behavior of objects. In object-oriented languages

[⋆] We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02) and "Recast: Evolution of Object-Oriented Applications" (SNF 2000-061655.00/1).

such as CLOS, Smalltalk or Ruby, classes themselves are first-class objects, and instances of so-called *metaclasses* [1,2,?,5,7]. In the same way that classes define the properties for their instances (objects), metaclasses implement the properties for their instances (classes). Examples of class properties are *Singleton*, *Final*, *Abstractness* . . . [8].

Treating classes as first-class objects and having metaclasses is important for two main reasons:

- *Uniformity and Control.* In a pure object-oriented language it is natural for classes to be instances of metaclasses. The uniformity defines metaclasses as the natural place to *specify* and *control* object creation and other class behavior.
- *Reuse of Class Behavior.* Since a metaclass is just like any other class, class behavior is reused and conventional reuse and decomposition techniques are applied to the metaclasses [8]. Hence the same techniques that are available for base classes (inheritance and overriding of methods, for example) are applicable at the meta level.

When a language has metaclasses, those metaclasses can be *implicit* or *explicit*. With *implicit* metaclasses the programmer cannot specify the metaclass for a class [9]. As such, implicit metaclasses successfully address the goal of "uniformity and control", but they fall short for achieving "reuse of class behavior". *Explicit metaclasses* avoid this limitation because the programmer can explicitly state from which metaclass his or her classes are instances [1,2,4,5].

Languages without explicit metaclasses suffer from the fact that class properties cannot be reused across classes, and that they cannot be combined. For example, every time one needs a class with the Singleton behavior, the same code needs to be implemented over and over again. With explicit metaclasses the singleton class property can be factored out to a Singleton metaclass, which can then be used to instantiate classes that exhibit the Singleton behavior. However languages with explicit metaclasses suffer from the fact that composition can be *unsafe* [2,10] or are based on *non-uniform* mechanisms *i.e.*, the meta-programmer cannot use the same composition mechanism used for programming at the base level than for programming at the meta level. This is clearly a problem, since metaclasses originate from the wish of uniformity in OOP (see Section 2).

To address these problems we propose to use the *general-purpose* object-oriented language feature *traits* [11]. Traits are composable units of behavior that close the large conceptual gap between a single method and a complete class. Our approach models class properties with traits, and uses trait composition to safely combine and reuse properties in metaclasses. Consequently, metaclass composition (like class composition) enjoys all the conceptual benefits of the traits composition model. In particular, composition conflicts that occur when composing two properties that do not quite fit together are detected automatically and the conflict resolution is *explicit* and under *control* of the composing entity.

As we will show in the rest of the paper, our solution supports the reuse of class properties, their safe and *automatic* composition with *explicit* conflict resolution, and the usage of the same mechanism (traits) for both the base and metalevel. As safety is a broad term we follow the definition of safe metaclass composition as defined in [12] and that we present in the following section. Now we start by identifying precisely

2 Explicit Metaclass Problems

Having explicit metaclasses promotes reuse but introduces several problems summarized in this section and detailed in the rest of the paper.

Unsafe Composition. Some approaches sacrifice the compatibility between the class and the metaclass level [2,10]. Unsafe metaclass composition means that code fragments applied to a class may break when used on another one due to the inheritance relationship between the metaclasses of the classes involved (See section 3).

Ad-Hoc and Non-Uniformity. There are some approaches that are specifically designed to avoid the compatibility problems raised in the first point. Their solutions, however, rely on *ad-hoc* composition mechanisms that are based on automatic code generation and dynamically changing the meta-metaclass [12]. Not only does this make it hard to understand the resulting code, it also leads to problems in case of conflicting properties and results in hierarchies that are fragile with respect to changes. Note that MetaclassTalk by using mixin composition at the metalevel is the only solution that solves this problem [15].

The solutions are not satisfactory from a conceptual point of view either, because the meta level (or meta meta level) does not employ object-oriented techniques (such as inheritance or instantiation) but *ad-hoc* mechanisms only applicable for metaclass composition. This breaks the fundamental idea of reflective programming that uses the *available* features of a language to define and control the behavior of the language itself [4].

Limited Composition. Other approaches used in the specific context of metaobjects use *chain of responsibility* [13] or composite metaobjects [14] to compose metaobjects. The first approach does not provide full control over the composition. The second approach forces the programmer to develop specific metaobjects to compose others, even when the reuse of these composite metaobjects is unclear.

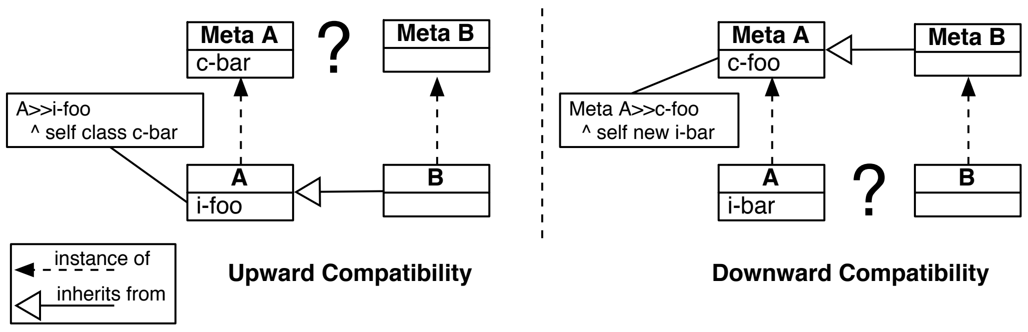

Fig. 1. Left: Upward compatibility - dependencies on the base level need to be addressed at the meta level. Right: Downward compatibility - dependencies on the meta level need to be addressed at the base level.

The ideal metaclass composition solution would make composition be *automatic*. However, as we will discuss in Section 11 a simple solution does not exist in a context where new properties can be defined and composed, and where their semantics can severely conflict. So the solution is a mechanism that is both safe and uniform *i.e.*, one that does not require the developer to make a paradigm shift and where the development of base-level applications and meta-level applications is the same.

3 Qualifying Composition

Offering explicit metaclasses is a way to reuse class properties but it also opens the door for *metaclass compatibility problems* [10]. This section defines criteria by which approaches that solve metaclass composition problems can be characterized and distinguished. We start by listing two criteria that were already identified in [12] (*upward*, *downward* compatibility and *per class property*), and then introduce three new ones that were not previously considered (*property composition*, *property application*, and *control of the composition*), but that qualify the problem in a more detailed way.

Upward Compatibility. The fact that classes are instances of other classes which define their behavior introduces hidden dependencies in the inheritance relationships between the classes and their metaclasses. Careless inheritance at one level (be it the class or metaclass level), can break inter-level communication. N. Bouraqadi et al. [12] refined the metaclass compatibility problems in two precise cases named *upward* and *downward* compatibility.

Let B be a subclass of 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.

Figure 1 left illustrates upward compatibility. When an instance of B receives the message i-foo, the message c-bar is sent to B. The composition of A and B is upward compatible, if B understands the message c-bar, *i.e.*, MetaB should implement it or somehow inherit it from MetaA.

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.*

Downward compatibility is illustrated in Figure 1 right. When B receives the message c-foo, the message i-bar is sent to a newly created instance of B. The composition of MetaA and MetaB is downward compatible, if that new instance of B understands the message i-bar, *i.e.*, B should implement it or somehow inherit it from A.

Definition. Metaclass composition is *safe* when it supports downward and upward compatibility.

Per Class Property. Different metaclass properties can be assigned to different classes in an inheritance hierarchy. Some systems such as NeoClasstalk and MetaClasstalk allow one to assign a property to a *single* class without it being inherited by its subclasses [12,15]. The authors of NeoClasstalk and MetaClasstalk, N. Bouraqadi et al. defined *class property propagation* as follows: "A property assigned to a class is automatically propagated to its subclass.". We name this criteria *per class property.* For example it is possible to define that a class is abstract and its subclasses are not abstract and this without having to redefine the property at the subclasses level.

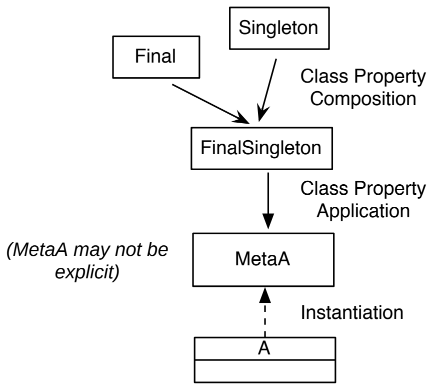
Singleton
Final
Class Property Composition
FinalSingleton
Class Property Application
(MetaA may not be explicit)
MetaA
Instantiation
A

Fig. 2. Property Composition and Property Application: two different stages in the process of reusing class properties.

Property Composition. One of the main motivations for having explicit metaclasses is to *combine class properties*, as shown in Figure 2, so that one class can for example be both a Singleton and Final. Hence a mechanism is needed that supports such property composition. This can be a general-purpose language mechanism such as multiple inheritance [4,5], mixin composition [15], chain of responsibility [13], or an *ad-hoc* mechanism such as generation of new classes and methods [12].

Property Application. Property application is the mechanism by which the composed properties are applied to classes. As shown in Figure 2 we distinguish the *composition* of properties from the *application* of a property to a specific class because some approaches employ different techniques for these two purposes. As an example, SOM uses ordinary multiple inheritance to compose class properties but it employs a combination of multiple inheritance and code generation to apply a class property to a class.

Control. The mechanism used to apply and combine class properties can be *implicit* or *explicit*. We call the mechanism *implicit* if the system automatically combines or applies the class properties and implicitly resolves conflicts in a way that may or may not be what the programmer intends. We call the mechanism *explicit* if the system gives the programmer explicit control over how the properties are combined and applied. In particular, the programmer should have *explicit control* over how conflicts are resolved. For many approaches, this is not the case because the composition of properties is based on a chain of responsibility which does not provide full control of the composition.

4 Traits in a Nutshell

Traits [11] are an extension of single inheritance with a similar purpose as mixins but avoiding their problems. Traits are essentially groups of methods that serve as building blocks for classes and are primitive units of code reuse. As such, they allow one to factor out common behavior and form an intermediate level of abstraction between single methods and complete classes. A trait consists of *provided methods* that implement its behavior, and of *required methods* that parameterize the provided behavior. Traits cannot specify any instance variables, and the methods provided by traits never directly access instance variables. Instead, required methods can be mapped to state when the trait is used by a class.

With traits, the behavior of a class is specified as the composition of traits and some *glue methods* that are implemented at the level of the class. These glue methods connect the traits together and can serve as accessor for the necessary state. The semantics of such a class is defined by the following three rules:

- *Class methods take precedence over trait methods.* This allows the glue methods defined in the class to override equally named methods provided by the traits.

- *Flattening property.* A non-overridden method in a trait has the same semantics as the same method implemented in the class.
- *Composition order is irrelevant.* All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated.

Because the composition order is irrelevant, a *conflict* arises if we combine two or more traits that provide identically named methods that do not originate from the same trait. Traits enforce explicit resolution of conflicts by implementing a glue method at the level of the class that overrides the conflicting methods, or by *method exclusion*, which allows one to exclude the conflicting method from all but one trait. In addition traits allow *method aliasing.* The programmer can introduce an additional name for a method provided by a trait to obtain access to a method that would otherwise be unreachable, for example, because it has been overridden. Traits can be composed from subtraits. The composition semantics is the same as explained above with the only difference being that the composite trait plays the role of the class.

5 Using Traits to Reuse and Compose Class Properties

Our approach is based on using traits to compose and reuse class properties within the traditional parallel inheritance schema proposed by Smalltalk (See Figure 8 left). Therefore our approach is safe *i.e.*, it supports downward and upward compatibility. But on top of that it promotes the reuse of class properties. Composition and application of class properties are based on trait composition, which gives the programmer explicit control in a uniform manner.

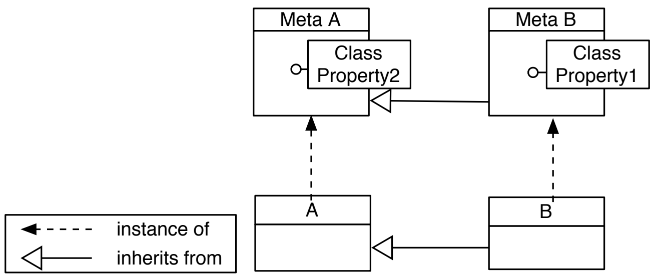

Fig. 3. Metaclasses are composed from traits representing class properties. Traits supports upward and downward compatibility.

We represent class properties as traits, which are then used to compose metaclasses as shown in Figure 3. Since traits have been fully implemented in the open-source Squeak Smalltalk environment [19], we implemented all the examples shown here in Squeak. During our refactoring of Squeak code we identified the following class properties: TAbstract, TSingleton, TRememberInstances, TCreator, and TFinal which we explain below. We start with a simple example illustrating how a class is composed by reusing a class property, then we look how the traditional Boolean hierarchy [8,12] is re-expressed with traits and finally Section 6 shows that traits provide a good basis to engineer the meta level.

5.1 Singleton

To represent the fact that a class is a Singleton, we define the trait TSingleton. This trait defines the following methods: default which returns the default instance, new which raises an error, and reset which invalidates the current singleton instance. It requires basicNew which returns a newly created instance [1] , and the methods uniqueInstance and uniqueInstance:. Note that these accessors methods are needed because traits cannot contain instance variables. Figure 4, left, shows the trait TSingleton.

```
Trait named: #TSingleton uses: {} category: 'Traits-Example'
TSingleton≫default
    self uniqueInstance isNil
        ifTrue: [self uniqueInstance: self basicNew].
    ↑ self uniqueInstance
TSingleton≫new
    self error: 'You should use default'
TSingleton≫reset
    self uniqueInstance: nil
```

As an example, suppose that we want to specify that a certain class WebServer is a Singleton. First of all we define the class WebServer in the traditional Smalltalk way as shown in 4. Then we specify at the metaclass level *i.e.*, in the class WebServer class, that the class is a Singleton by specifying that the class is composed from the trait TSingleton. The metaclass defines state needed to keep an instance around, under the form of the instance variable uniqueInstance. It also defines two glue methods uniqueInstance and uniqueInstance: as accessors methods for the instance variable uniqueInstance. These two glue methods fulfill the required methods with the same name of the trait TSingleton. Note that the required method basicNew is provided by the class Behavior, of which WebServer class, is an indirect subclass (see Figure 4, right).

[1] Using basicNew is the traditional way to implement Singleton in Smalltalk when we want to forbid the use of the new method [20]. basicNew allocates objects without initializing them. It is a Smalltalk idiom to never override methods starting with 'basic' names.

```
Object subclass: #WebServer
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Traits-Example'

WebServer class uses: {TSingleton }
    instanceVariableNames: 'uniqueInstance'

WebServer class>>uniqueInstance
      ↑ uniqueInstance

WebServer class>>uniqueInstance: anObject
      uniqueInstance := anObject
```

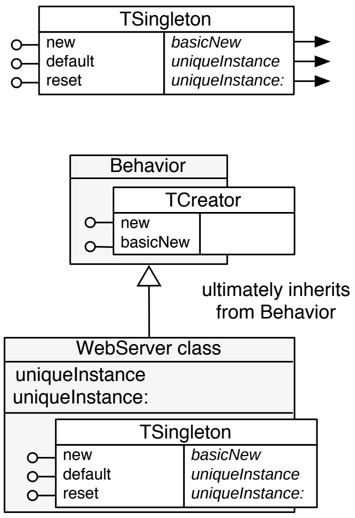

Fig. 4. Left. The trait TSingleton. Right. The class Behavior, the root of metaclasses in Smalltalk, is composed from the trait TCreator and as such provides the method basicNew.

5.2 The Boolean Hierarchy Revisited

The Smalltalk Boolean hierarchy consists of the abstract class Boolean, that has two subclasses True and False that are singleton classes. Traits allow the boolean hierarchy to be refactored as shown in Figure 5. Note that the refactored solution is backwards compatible with the idioms existing in the current Smalltalk implementation and literature [20]. So we assume that a method basicNew is defined on the class Behavior that can always be invoked to allocate instances and that should not be overridden.

Boolean. The class Boolean is an abstract class, so we compose its class Boolean class from the trait TAbstract.

```
Trait named: #TAbstract uses: {} category: 'Traits-Example'
TAbstract≫new
    self error: 'Abstract class. You cannot create instances'
TAbstract≫new: size
    self error: 'Abstract class. You cannot create instances'
```

False and True. The classes False and True are Singletons so their classes False class and True class are composed from the trait TSingleton which is then reused in the two classes.

As mentioned above, the trait TSingleton requires the methods basicNew, uniqueInstance, and uniqueInstance:. Therefore the class False class (resp. True class) has to define an instance variable uniqueInstance and the two associate accessors methods uniqueInstance and uniqueInstance:. Note that the method basicNew does not have to be redefined locally in the class False or True class as it is inherited ultimately from the class Behavior, the inheritance root of the metaclasses [9] (see Figure 5 right). This example shows that class properties are reused over different classes and that metaclasses are composed from different properties.

```
False class
    uses: {TSingleton }
    instanceVariableNames: 'uniqueInstance'
False class≫uniqueInstance
    ↑ uniqueInstance
False class≫uniqueInstance: anObject
    uniqueInstance := anObject
```

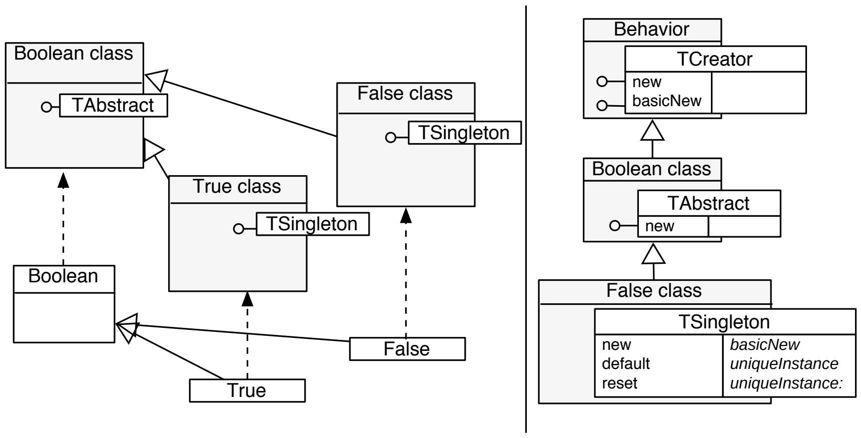

Fig. 5. Left: Boolean hierarchy refactored with traits. Right: The complete picture for the Boolean hierarchy solution.

6 Engineering The Meta Level

So far we presented simple examples that show how traits are well-suited to model class properties, which can then be combined or applied to arbitrary classes. In this section, we show that traits also allow more fine-grained architectures of class properties. We also want to stress that the techniques used here at the meta level are exactly the same as those used at the base level. As such, traits provide a uniform model.

Since many of these properties are related to instance creation, and we perform our experiments in Smalltalk, we first clarify the basic instance creation concept of Smalltalk. In Smalltalk, creation of a new instance involves two different methods, namely basicNew and new [2]. The method basicNew is a low-

[2] Note that there are also the methods basicNew: and new:, which are used to create

level primitive which simply allocates a new instance of the receiver class. The method new stands at a conceptually higher level and its purpose is to return a usable instance of the receiver class. For most classes, new therefore calls basicNew to obtain a new instance and then initializes it with reasonable default values.

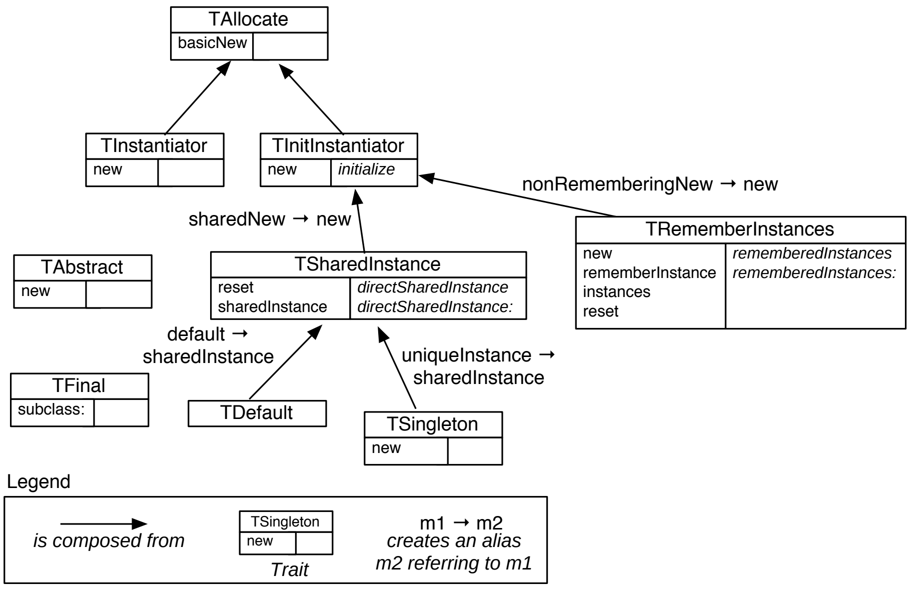

Fig. 6. A fine-grained architecture of class properties based on traits

6.1 Class Properties

Figure 6 gives an overview of the class properties we identified (see Section 10 for a deeper discussion). Note that all of these properties are traits, and that they are therefore composed using trait composition.

Allocation. As indicated by its name, the trait TAllocator provides the behavior to allocate new instances. In our case, this is the standard Smalltalk basicNew method, but of course we could also create another trait with an alternative allocation strategy.

Instantiation. The traits TInstantiator and TInitInstantiator are two class properties for instance creation. The trait TInstantiator uses the trait TAllocator and implements the method new in the traditional Smalltalk manner, which means that it does not initialize the newly created instance. The trait TInitInstantiator uses the trait TAllocator. However, as suggested by its name, it actually ini-

objects with indexed fields (*i.e.,* arrays). For sake of simplicity, we do not take these methods into account here.

tializes the newly created instance by calling the method initialize before the instance is returned.

```
TInstantiator≫new
    ↑self basicNew
TInitInstantiator≫new
    ↑self basicNew initialize
```

Note that the method initialize is called on the new instance, which means that the requirement for initialize in the trait TInitInstantiatior is actually a requirement for the instance side.

Remembering Instances. The trait TRememberInstances represents an instance creation property that remembers all the instances created by a class. It uses the trait TInitInstantiator and aliases the method new of the traits TInitInstantiator which is then available as nonRememberingNew. This aliasing allows one to access the original new method of the trait TInitInstantiator while leaving the option to override the method new in the trait TInitInstantiator. It requires the methods rememberedInstances and rememberedInstances: to access a collection storing the created instances. Then, it implements the methods new, rememberInstance:, instances, and reset as follows:

```
TRememberInstances≫new
    ↑ self rememberInstance: self nonRememberingNew
TRememberInstances≫rememberInstance: anObject
    ↑ self instances add: anObject
TRememberInstances≫instances
    self rememberedInstances ifNil: [self reset].
    ↑ self rememberedInstances
TRememberInstances≫reset
    self rememberedInstances: IdentitySet new
```

Note that another implementation could be to define the methods reset and rememberedInstances: as trait requirements. This would leave the class with the option to use other implementations for keeping track of the created instances.

Default and Singleton. The traits TDefault and TSingleton implement the class properties corresponding to the Default Instance and Singleton design patterns. Whereas a Singleton can only have one single instance, a class adhering to the Default Instance pattern has one default instance but can also have an arbitrary number of other instances.

Since these two properties are very similar, we factored out the common code into the trait TSharedInstance. To get the basic instantiation behavior, this trait uses the property TInitInstantiator and again applies an alias to ensure that the method new is available under the name sharedNew. Then, it implements the methods reset and sharedInstance as follows:

```
TSharedInstance≫reset
    self directSharedInstance: self sharedNew.
TSharedInstance≫sharedInstance
    self directSharedInstance ifNil: [self reset].
    ↑ self directSharedInstance.
```

The property TDefault is then defined as an extension of the trait TSharedInstance that simply introduces the alias default for the method sharedInstance. Similarly, the property TSingleton introduces the alias uniqueInstance for the same method. In addition, TSingleton overrides the method new so that it cannot be used to create a new instance:

```
TSingleton≫new
    self error: 'Cannot create new instances of a Singleton.
        Use uniqueInstance instead'.
```

Another useful class property popularized by Java is the class property TFinal which ensures that a class cannot have subclasses. In Smalltalk, this is achieved by overriding the message subclass: [3]. Note that unlike all the other properties presented in this section, TFinal is not concerned with instance creation and therefore is entirely independent of the other properties. In Section 10 we discuss the relevance of the class properties we presented.

6.2 Advantages for the Programmer

Having an architecture of class properties has many advantages for a programmer. Whenever a new class needs to be created, a choice can be made regarding the creation of instances, and whether or not the class should be final. Besides having the obvious advantage of avoiding code duplication, it also makes the design much more explicit and therefore facilitates understandability of the class. The level of abstraction of the trait design is at the right level: the traits correspond to the class properties, and the class properties can be combined into metaclasses.

In addition, factoring out the properties in such a fine-grained way still gives the user a lot of control about some crucial parts of the system. Suppose for example that at first we would have decided to use the trait TInitInstantiator as the basis for all the other instance creation properties. If later on, we would decide to comply to the Smalltalk standard to create uninitialized instances by default, then we could make this change without modifying any of the involved methods. We would just need to make sure that the traits TRememberInstances

[3] In reality, the method to create a subclass takes more arguments but this is not relevant here

and TSharedInstance use the trait TInstantiator instead of TInitInstantiator.

Explicit Composition Control Power. By providing several different properties that are all related to instance creation behavior, this example also shows why it is so important to have explicit control over composition and application of class properties. In our example, there are many different properties which essentially introduce variants of the method new, and therefore, combining these properties typically leads to conflicts that can only be resolved in a *semantically* correct manner if the user has explicit control over the composition. In case of traits, this is ensured by allowing partially ordered compositions, exclusions, and aliases.

As an example, imagine that we want to combine the properties TDefault and TRememberInstances to get a property that allows both a default instance and also remembers all its instances. With our trait-based approach, we do this by creating a new trait TDefaultAndRememberInstances which uses TRememberInstances and TDefault as follows:

```
Trait named: TDefaultAndRememberInstances
    uses: {  TDefault @ {#defaultReset → #reset}.
             TRememberInstances − {#new}
                 @ {#storeNew → #new.
                     #storeReset → #reset}}

TDefaultAndRememberInstances≫sharedNew
    ↑self storeNew

TDefaultAndRememberInstances≫reset
    self storeReset.
    self defaultReset
```

Since both traits provide a method new, we exclude this method from the trait TRememberInstances when it is composed. As a consequence the trait contains the new method provided by TDefault, which uses sharedNew to create a new instance. Since we want to make sure that each new instance is also stored, we override sharedNew so that it calls storeNew, which is an alias for the new method provided by TRememberInstances.

Because the method reset is also provided by both traits, we use aliasing to make sure that we can access the conflicting methods. Then, we resolve the conflict by overriding the method reset so that it first removes the stored instances (by calling storeReset) and then creates a new default instance (by calling defaultReset). Note that the newly created instance will be remembered as the default instance and will also be stored in the collection with all the instances of the class.

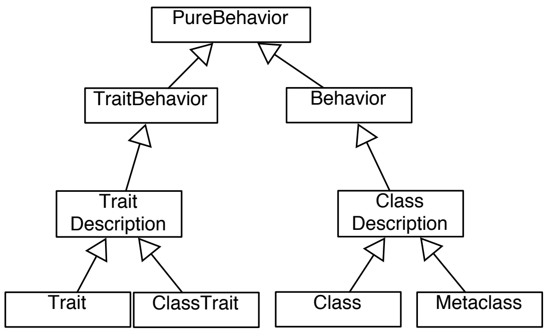

Fig. 7. The hierarchy of the new Smalltalk kernel supporting traits.

7 A New Kernel

In this section we present briefly the key implementation aspects of the new Smalltalk kernel that is bootstrapped with traits. The Figure 7 shows the new class hierarchy that we obtain. As the new kernel with traits is an extension of the traditional Smalltalk kernel, we get the traditional classes: Behavior, ClassDescription, Metaclass, and Class which now deal with the fact that a class may be composed of traits.

To model traits we then follow the previous design of the kernel and mimic the classes Behavior, ClassDescription and Class. Three classes TraitBehavior, TraitDescription, and Trait are introduced. Trait represents a trait and is applied to both the class and instance side. In addition the class Behavior, root of the instantiation graph, uses two important traits: TInstantiator and TInitInstantiator as presented in previous section. The class Behavior in Smalltalk also defines information about the state and behavior related to superclass and instance variables (format). As this is not needed for traits, we introduce a new abstract superclass PureBehavior, which factors out the common code between Behavior and TraitBehavior.

Some Traits. The traits we identified and used in this new kernel are not really remarkable in the sense of new MOP entries. In fact we mainly use traits to reuse code between the classes TraitsDescription and ClassDescription for the following reasons.

- The new kernel is based on the traditional Smalltalk kernel, which uses inheritance as the primary reuse mechanism. A lot of polymorphic methods are used among the classes Behavior and ClassDescription, and the classes Class and Metaclass. As a consequence, there is not much need for introducing traits to share this functionality.
- In the traditional Smalltalk kernel, the class Behavior only defines the minimal state and behavior to support classes as run-time entities. For example Behavior does not define the notion of named instance variables but just

knows the format of the instances the class will create in terms of the number and kind of instance variables [9]. Support for named instance variables is (amongst other things) implemented by ClassDescription, a subclass of the class Behavior. Because of this inheritance structure, some code that is implemented in ClassDescription (and uses state defined in ClassDescription) cannot and should not be reused by pushing it up to PureBehavior. Instead, we use traits to share behavior between ClassDescription and TraitDescription.

The class PureBehavior uses the following traits:

- TBasicCompile supports the compilation and decompilation of methods in a class.
- TTestingSelector supports the testing of selectors of methods (*e.g.*, canUnderstand:, ...).
- TCompiledMethodAccess supports the access into the method dictionary and source code access.
- TMethodIterating supports the iteration over compiled methods.

The class Behavior uses the following traits:

- TInstantiator and TInitInstantiator implement the creation of objects as described in the previous section.
- TFamilyAccess supports the access and enumeration of superclasses and subclasses.
- TInstanceEnumerator supports the enumeration of instances of the class.
- TMethodTesting supports the querying of methods.

The traits resulting from the decomposition of the classes PureBehavior and Behavior are not currently used by any other classes. In contrast, the following traits are used to share behavior between the two classes TraitsDescription and ClassDescription.

- TClassComment supports the management of comments.
- TMethodDictionaryManagement supports the management of methods categories.
- TOrganization supports how methods are sorted into method categories.
- TCodeReformatting supports reformatting of source code.
- TCodeFileOut supports filing-out (saving) of classes.
- TBehaviorCopy supports the copying of methods and their organization.
- TOrganizedCompilation supports the compilation of methods within the context of categories.

It should be noted that in the Smalltalk metaclass kernel, identifying traits that can be reused independently of each others is difficult because the behavior of the kernel is based on inheritance and the code was tightly coupled.

	up	down	per class	application	composition	control
Smalltalk	Yes	Yes	No	No	No	No
CLOS	Yes	Yes	No	multiple inheritance	multiple inheritance	explicit + linearization
SOM	Yes	No	No	multiple inheritance + code generation	multiple inheritance	implicit
NeoClasstalk	Yes	Yes	Yes	inheritance + generation	inheritance + code generation	implicit
MetaclassTalk	Yes	Yes	Yes	inheritance	mixin composition	mixin linearization
Traits	Yes	Yes	No	trait composition	trait composition	explicit

Table 1
Comparison of the models from Section 8 on how they handle the composition problems described in Section 2.

8 Related Work

This section shows how the main approaches that support explicit metaclasses address the problems described in Section 2. We also discuss the solution offered by Smalltalk (although it has implicit metaclasses) since it forms the basis for the NeoClasstalk solution and our own solution. Table 1 summarizes the comparison of these approaches. Note that the table shows the influence of the CLOS approach based on multiple inheritance to support metaclass composition in SOM.

8.1 Metaclass Composition

Smalltalk. In Smalltalk (and more recently in Ruby), metaclasses are *implicit* and created *automatically* when a class is created [9]. Each class is the sole instance of its associated metaclass. This way the two hierarchies are parallel (see Figure 8 left). Hence the architecture is safe as it addresses compatibility issues but completely prevents class property reuse between several hierarchies.

CLOS. CLOS's approach could be summarized as "do it yourself". Indeed by default in CLOS, a class and its subclasses must be instances of the same metaclass, prohibiting classes in the same hierarchy from having different class properties. For example, in Figure 8 right, class B has by default the same metaclass as its superclass A, and this cannot be changed. So class B always has the same class properties as class A. Note that since CLOS has explicit metaclasses, multiple inheritance can be used for composing class properties. For example, in the context described by Figure 2 it is possible to use multiple inheritance to explicitly combine the two properties Final and Singleton expressed as metaclasses into a new class SingletonFinal. Note that such an

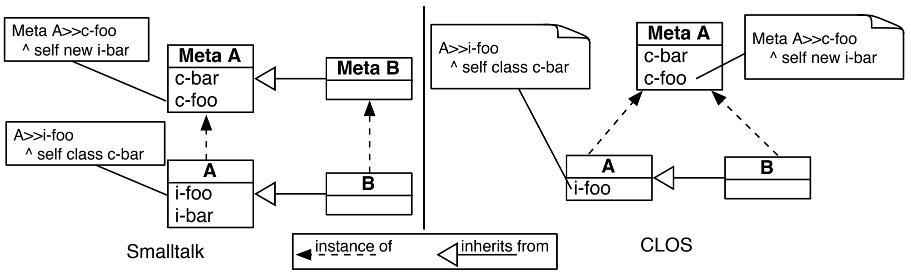

Fig. 8. Left: Smalltalk addresses compatibility issues by preventing reuse using implicit metaclasses and parallel hierarchies. Right: By default CLOS addresses compatibility issues by preventing subclasses to have different metaclasses than their superclasses.

implementation suffers from the same problems as multiple inheritance based on linearization occurring at the base level [16].

The general CLOS rule that a class and its subclasses must be instances of the same metaclass can be circumvented using CLOS's metaobject protocol (MOP). Indeed, the generic function validate-superclass [4] offers a metaprogrammer the possibility to specify that a class and its subclasses can be instances of different classes. However, this comes at a very high price because the CLOS MOP does not provide predefined strategies for avoiding compatibility problems or for dealing with possible conflicts. Hence the semantics of the composition has to be implemented manually, a far from trivial undertaking.

This means that by default CLOS is upward and downward compatible but it prevents usage of different metaclasses within an inheritance hierarchy and reuse of class properties. Both the composition of class properties and the application of properties are done with multiple inheritance. The control of the composition is explicit, because the user has to use multiple inheritance to create a new metaclass. However, since multiple inheritance in CLOS uses implicit linearization, the well-known problems associated with this form of conflict resolution also apply to the meta level [16].

SOM. The solution proposed by SOM (System Object Model) [7] is based on the automatic generation of *derived metaclasses*, that inherit multiply from the metaclasses to compose class properties. When at compile time a class is specified to be an instance of a certain metaclass, SOM automatically determines whether upward compatibility is ensured and if necessary creates a derived metaclass. In Figure 9 left, the class B (originally an instance of MetaB), inheriting from class A (instance of MetaA) finally becomes an instance of a derived metaclass inheriting from MetaA and MetaB. Note that SOM ensures that the existing metaclass MetaB takes precedence over MetaA in case of multiple inheritance ambiguities (since B is a subclass of A).

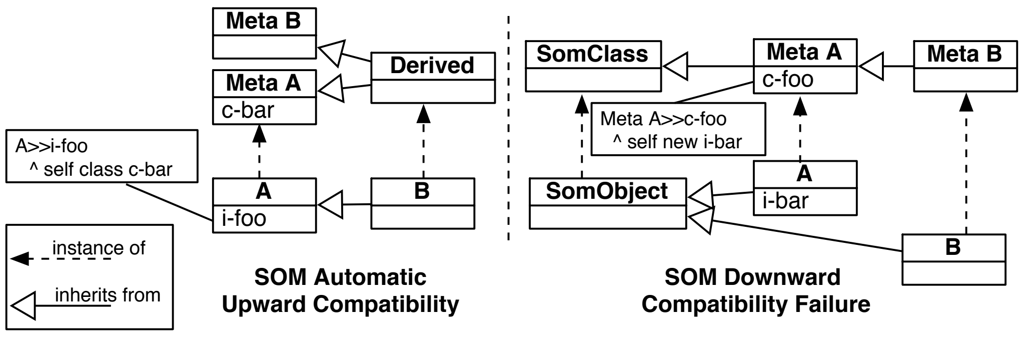

Fig. 9. Left: SOM supports upward compatibility by automatically deriving new metaclasses and changing the class of the inheriting class B. Right: SOM downward compatibility failure example.

While SOM supports upward compatibility as shown in Figure 9 left, it does not support downward compatibility [12] as shown in Figure 9 right. When the class B receives the c-foo message, a run-time error will occur because its instances do not understand the i-bar message. However, in SOM, contrary to CLOS, two distinct classes need not have the same metaclass. But as in CLOS, the composition of class properties is based on multiple inheritance. The application of a class property is done by a combination of multiple inheritance and automatic class generation. This happens at compile time, and the programmer has no explicit control over how possible conflicts are resolved.

NeoClasstalk. NeoClasstalk's approach is interesting since it supports both downward and upward compatibility and enables class property reuse between different hierarchies [17,12,18]. NeoClasstalk uses two techniques to accomplish this: *dynamic change* of classes and the composition of metaclasses by *code generation*. It generalizes the parallel inheritance solution of Smalltalk by enabling class properties reuse, but it also introduces some problems of its own that we discuss in detail after explaining the basic principles.

NeoClasstalk allows properties to be assigned to classes. Figure 10 shows what happens when assigning a property to Meta B. B inherits from class A and is an instance of the class Meta B before the new property is assigned to Meta B. When assigning the property, the system automatically creates a new metaclass Property m + Meta B (called a *property metaclass*), which inherits from the metaclass Meta B and defines the property code. It then *changes the class* of B to be that newly created metaclass. NeoClasstalk supports also per class property, *i.e.*, a property added to a class does not get automatically propagated to its subclasses.

To be able to reuse the property classes, NeoClasstalk stores the class properties in strings on methods of so-called *meta-metaclasses*. The actual metaclasses are then generated from these strings, as shown for our example in Figure 10. For example, the Property m represented by a meta-metaclass is

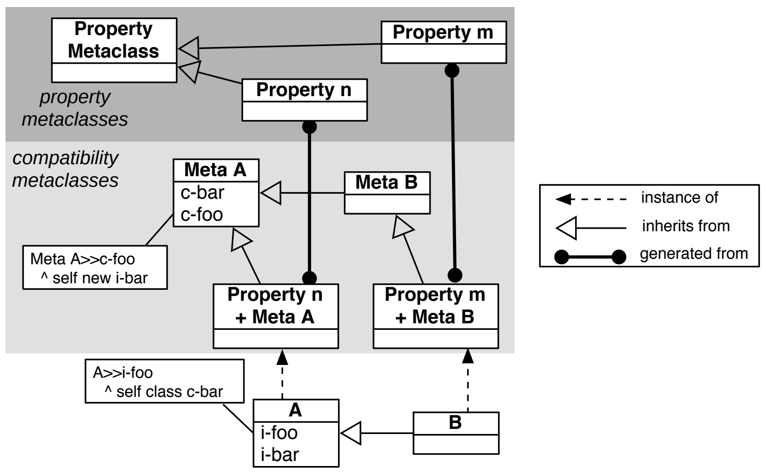

Fig. 10. Assigning the property m to class Meta B and property n to the class Meta A in NeoClasstalk. The light grey area denotes the metaclass area. The dark grey area is the realm of the class properties.

used to generate a new metaclass named Property m + Meta B from the metaclass Meta B and the Property m.

Besides the intrinsic complexity of NeoClasstalk's approach, it has the following drawbacks:

Dynamic class creation and dynamic change of class. The approach relies on the dynamic creation of classes and the dynamic changing of classes. It induces a complex management of meta-metaclass changes that should be propagated to the generated instances. Moreover as programming at the meta meta level is based on manipulating the strings that represent bodies of methods of metaclasses, it is not the same as programming at the metaclass or the base level. Basically, despite the name, the property metaclasses are not really at the meta-metaclass level, but merely storage holders for strings. The relation between the meta-metaclass level and the metaclass level is therefore not instantiation, as one would expect, but code generation. This breaks the uniformity of the model.

Ad-hoc and Implicit Composition. Property metaclasses are composed by code generation and applied implicitly by defining them in an inheritance chain. The composition is based on the assumption that a metaclass is designed to be plugged in this inheritance chain and that other composed behavior can be reached via super invocations. The composite metaclass has only limited control over the composed behavior as it can only invoke overridden behavior but does not have the full composition control.

As a summary, NeoClasstalk provides both downward and upward compatibility, and it allows one to assign class properties on a per-class basis. The composition of class properties is implicit and based on code generation and chain of responsibility. The application of class properties is based on dynamic class changes and code generation.

MetaclassTalk. MetaclassTalk follows the architecture of NeoClasstalk by offering compatibility and property metaclasses. MetaclassTalk uses mixin composition to compose metaclass properties [15]. This experiment makes MetaclassTalk the closest model to our own approach as it supports both downward and upward compatibility while allowing the reuse of class properties. However MetaclassTalk composition is based on mixin linearization. As such it has the same problems as the ones we present in [11]: the composite entities do not have the full control of the composition, and the glue code is spread over multiple classes. These problems are solved by traits.

8.2 Metaobjects

Other approaches such as CodA [23], Moostrap [13], Iguana/J [24] support the composition and reuse of metaobjects. Such a composition is often based on chain of responsibility [13] *i.e.*, a metaobject is designed to be composed in a chain of metaobjects by invoking the overridden functionality. The problem with chain of responsibility is that it forces all the metaobjects to follow a certain architecture. It more importantly gives the composing metaobject only a very limited control over the composition: it can invoke the rest or do nothing. In contrast, traits composition is automatic when there is no conflict, and when conflicts arise, the composing metaclass has complete control over all the composed class properties.

The authors of Guarana [14] and Reflex [25], introduce *composite metaobjects i.e.*, a metaobject that define the composition semantics of several metobjects. This approach works well for coarse-grained composition, such as for making changes to the message passing semantics (broadcast, concurrent dispatch, or remote invocations). However, it is too heavyweight to compose class properties, since it would force the developer to define an explicit *composite metaclass* for all simple conflicts whose reuse is even questionable.

CodA [23] structures the meta-level architecture around several metaobjects responsible for the different actions. However it raises the issue of compatibility between all the metaobjects associated to a given object. The solution is to manually define a semantically coherent configuration of metaobjects implementing the desired semantics [23]. This solution shows again that there is no magic and that composing operations with conflicting semantics cannot be achieved in an automatic fashion.

9 Advantages and Disadvantages

Advantages. Traits support the decomposition of class properties as reusable units of behavior. Since metaclasses are composed of traits and the model is based on the parallel hierarchy of Smalltalk, it is upward and downward compatible and supports the reuse of class properties across different hierarchies.

In addition the proposed model is uniform with respect to the concepts used at the base level and the meta level (like CLOS). Both levels use the same concepts (traits and inheritance). Furthermore, the model is simple, and there is no need for on-the-fly code generation (as in SOM or NeoClasstalk) or for dynamic changes to classes (as in NeoClasstalk).

Class properties can be composed of traits that represent those properties. The application of the properties to an actual metaclass is accomplished by using the appropriate composite trait in the metaclass definition. The composing metaclass has *complete* control of the composition, and possible conflicts are resolved *explicitly* when the property is applied to a metaclass.

Having explicit control over the composition is especially important because it allows a programmer to freely adapt the behavior of the composite metaclass and to compose class properties that may not quite fit together. This means that our approach lets system designers ship their class hierarchies together with a set of *prefabricated* class properties in the form of traits, which can then be used and combined by the programmers. In case some class properties built by different vendors do not quite fit together, the traits model not only indicates the resulting conflicts, but also provides the programmer with the necessary means to resolve the conflicts to achieve the expected semantics.

Disadvantages. Glue methods and state have to be redefined in the metaclass where a property is applied. For example, the instance variable `uniqueInstance` and the two accessors methods have to be defined in all classes that implement a Singleton. We consider this to be a limit of the traits model and the price to pay to have the minimal mechanism supporting traits composition. Introducing state into traits would solve this but would introduce other problems, such as the well-known *diamond problem* of multiple inheritance [21], where state gets inherited through different paths.

It may happen that instance variables defined in a superclass are not necessary in the subclasses. For example, if the superclass implements a Singleton and the subclasses do not, then the instance variable that holds the Singleton instance as well as the methods to access it will be inherited by the subclasses. However, this problem is not due to traits by itself but is a result of using the inheritance mechanism in general. Table 1 compares the approaches.

10 Class Properties

Traits let us decide if a given functionality is defined as a trait or as a class. When defining functionality as a trait we automatically offer the possibility for future classes to use the identified behavior. One might wonder why we have so few class properties. First of all we chose to reengineer the current implementation of Squeak and not to design a new metaobject protocol. In this article we present the main *class properties* that we identified during our implementation, and we did not invent new ones. Secondly, we deliberately took heavily conflicting class properties, so that we could could clearly show the conflict resolution advantage of traits. Composing non-conflicting properties is trivial. Thirdly, other important efforts to build metaclass libraries, such as SOM [7], present nearly the same set of class properties.

Another point to consider is the role of the classes in the context of a metaobject protocol [4]; we believe that a lot of class properties identified in [8] are due to the fact that the classes were the single entry point in their MOP, while certain responsibilities are definitely the responsibilities of other meta-entities such as methods. It is also out the scope of this paper to present a new metaobject protocol based on traits, even if this is definitely future work.

Class Property Propagation Our approach does not support per class property because we did not want to change the class creation protocol of Smalltalk-80. However, there is nothing in our approach that prevents us to support per class property in a similar way than the compatibility model does [12,15]. Figure 11 shows that the intermediate metaclasses Boolean class + Abstract, False class + Singleton, and True class + Singleton are composed of traits and that properties such abstractness of the class Boolean, are not propagated to its subclasses. Note that metaclasses such as Boolean class could also be composed of traits if the property have to be propagated to the subclasses.

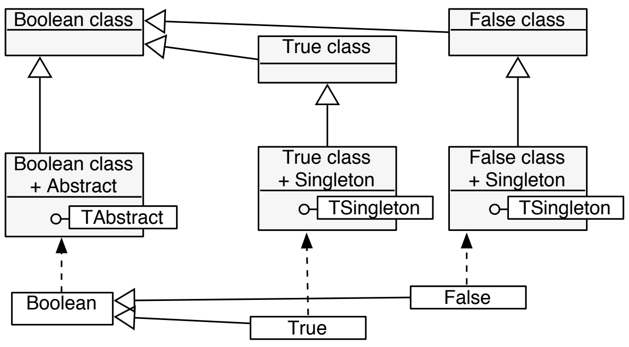

Fig. 11. Controlling class property propagation and trait composition.

11 Automatic Composition

An important difference between traits and most other approaches lies in the fact that traits have automatic conflict *detection* but expect *explicit* conflict resolution controlled by the programmer. Most approaches aim for a completely automatic resolution of conflicts, where possible conflicts are resolved according to some automatic scheme. If such schemes would work in all possible situations and in such a way that the programmer can easily foresee the result of a composition, developing software would be much easier. However, fully automatic resolution of conflicts is no panacea. It is only trivial when the composed semantics are orthogonal, so that conflicts can simply not occur. But it becomes extremely complex or even impossible when the semantics overlaps, which is the case for class properties. Looking in other areas such as multiple inheritance conflicts resolution we see that techniques based on automatic linearization techniques are not always satisfying [16] and often lead to unpredictable method invocations. The same applies here.

Nowadays the problems of composition of services or overlapping aspects is difficult and nearly impossible without the use of meta-data. For example Kienzle and Guerraoui demonstrates that trying to automatically compose transactions with other simpler aspects such as notification is doomed to failure [22]. In a similar vein authentication and encryption composition can only be a success when the encryption is invoked first, but authentication should take precedence over persistency and transactions.

Note that in the context of metaclass composition, the set of metaclass behavior is not predefined and fixed, as such it is possible to load a package in which another meta-programmer has developed new class properties with sensible composition exigence. Therefore any clever composition engine based on meta-information would have to deal with the openness of the set of class properties. Our solution, based on traits, differs from the other approaches since trait composition is automatic as long as there are no conflicts. Conflicts are detected automatically. When there is a conflict, then the traits model offers mechanisms to solve the conflict. This contrasts with the approaches that use an automatic scheme to handle conflicts.

12 Conclusion and Future Work

The need to reuse class properties led to meta-level architectures based on explicit metaclasses [1,2]. While offering reuse of class properties, such models introduced metaclass composition problems [10]. Different approaches exist that try to solve metaclass compositions problems, based on multiple inheritance, code generation or automatically changing metaclasses [5,12,15]. However, the definition, the composition and the application of the class property were not controllable by the developer or meta programmer.

Our solution models class properties with *traits* (first class groups of methods), and uses trait composition to safely combine and reuse them. Using traits to compose class properties first of all solves the metaclass composition problems (upward and downward compatibility is ensured) while supporting the reuse of class properties. In addition, composition and conflict resolution are *explicit* and under *control* of the composing entity. Thirdly, traits is a general-purpose composition mechanism for object-oriented languages that we have already applied successfully at the base level (for example to refactor the Smalltalk collection hierarchy [26]).

We implemented all the examples shown in this article using the Squeak implementation of traits and we started to refactor the kernel of Squeak using traits. Our next step is to use traits to define a new metaobject protocol for Smalltalk.

Acknowledgments. We also like to thank Eric Tanter, Noury Bouraqadi and the anonymous reviewers for their valuable comments and discussions.

References

[1] D. Ingalls, The Smalltalk-76 programming system design and implementation, in: POPL'76, 1976, pp. 9–16.

[2] P. Cointe, Metaclasses are first class: the objvlisp model, in: OOPSLA '87, 1987, pp. 156–167.

[3] J.-P. Briot, P. Cointe, Programming with explicit metaclasses in Smalltalk-80, in: OOPSLA '89, 1989, pp. 419–432.

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

[5] S. Danforth, I. R. Forman, Derived metaclass in SOM, in: TOOLS EUROPE '94, 1994, pp. 63–73.

[6] I. R. Forman, S. Danforth, H. Madduri, Composition of before/after metaclasses in SOM, in: OOPSLA '94, 1994, pp. 427–439.

[7] I. R. Forman, S. Danforth, Putting Metaclasses to Work: A New Dimension in Object-Oriented Programming, Addison-Wesley, 1999.

[8] T. Ledoux, P. Cointe, Explicit metaclasses as a tool for improving the design of class libraries, in: ISOTAS '96, LNCS 1049, Springer Verlag, 1996, pp. 38–55.

[9] A. Goldberg, D. Robson, Smalltalk-80: The Language, Addison Wesley, 1989.

[10] N. Graube, Metaclass compatibility, in: OOPSLA '89, 1989, pp. 305–316.

[11] N. Schärli, S. Ducasse, O. Nierstrasz, A. Black, Traits: Composable units of behavior, in: ECOOP 2003, LNCS 2743, Springer Verlag, 2003, pp. 248–274.

[12] N. M. N. Bouraqadi-Saadani, T. Ledoux, F. Rivard, Safe metaclass programming, in: OOPSLA '98, 1998, pp. 84–96.

[13] P. Mulet, J. Malenfant, P. Cointe, Towards a methodology for explicit composition of metaobjects, in: OOPSLA '95, 1995, pp. 316–330.

[14] A. Oliva, L. E. Buzato, The design and implementation of guarana, in: USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99), 1999.

[15] N. Bouraqadi, Safe metaclass composition using mixin-based inheritance, Journal of Computer Languages, Systems and Structures 30 (1-2) (2004) 49–61.

[16] R. Ducournau, M. Habib, M. Huchard, M. Mugnier, Monotonic conflict resolution mechanisms for inheritance, in: OOPSLA '92, 1992, pp. 16–24.

[17] F. Rivard, évolution du comportement des objets dans les langages à classes réflexifs, Ph.D. thesis, Ecole des Mines de Nantes, Université de Nantes, France (1997).

[18] S. Ducasse, Evaluating message passing control techniques in Smalltalk, Journal of Object-Oriented Programming (JOOP) 12 (6) (1999) 39–44.

[19] 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: OOPSLA '97, 1997, pp. 318–326.

[20] S. R. Alpert, K. Brown, B. Woolf, The Design Patterns Smalltalk Companion, Addison Wesley, 1998.

[21] A. Snyder, Inheritance and the development of encapsulated software systems, in: Research Directions in Object-Oriented Programming, MIT Press, 1987, pp. 165–188.

[22] J. Kienzle, R. Guerraoui, Aop: Does it make sense? the case of concurrency and failures, in: ECOOP '2002, LNCS 2374, Springer Verlag, 2002.

[23] J. McAffer, Meta-level programming with coda, in: ECOOP '95, LNCS 952, Springer, 1995, pp. 190–214.

[24] B. Redmond, V. Cahill, Supporting unanticipated dynamic adaptation of application behaviour, in: ECOOP 2002, LNCS 2374, Springer, 2002, pp. 205–230.

[25] E. Tanter, N. Bouraqadi, J. Noye, Reflex — towards an open reflective extension of java, in: International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, LNCS 2192, Springer, 2001, pp. 25–43.

[26] A. P. Black, N. Schärli, S. Ducasse, Applying traits to the Smalltalk collection hierarchy, in: OOPSLA '03, 2003, pp. 47–64.