Traits: Composable Units of Behavior

Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz
and Andrew Black

Department of Computer Science and Engineering
OGI School of Science & Engineering
Oregon Health & Science University
20000 NW Walker Road
Beaverton, OR 97006-8921 USA

Technical Report Number CSE 02-012
25th November 2002

A version of this paper has been submitted to the 2003
European Conference on Object-Oriented Programming (ECOOP).
If accepted, the paper will appear in the proceedings of ECOOP 2003,
and subsequent bibliographic citations should
refer to the conference proceeding

Traits: Composable Units of Behaviour*

Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black

Software Composition Group, University of Bern, Switzerland
OGI School of Science & Engineering, Oregon Health and Science University
{schaerli, ducasse, oscar}@iam.unibe.ch, black@cse.ogi.edu

Abstract. Inheritance is the fundamental reuse mechanism in object-oriented programming languages; its most prominent variants are single inheritance, multiple inheritance, and mixin inheritance. In the first part of this paper, we identify and illustrate the conceptual and practical reusability problems that arise with these forms of inheritance. We then present a simple compositional model for structuring object-oriented programs, which we call *traits*. Traits are essentially groups of methods that serve as building blocks for classes and are primitive units of code reuse. In this model, classes are *composed* from a set of traits by specifying *glue code* that connects the traits together and accesses the necessary state. We demonstrate how traits overcome the problems arising with the different variants of inheritance, we discuss how traits can be implemented effectively, and we summarize our experience applying traits to refactor an existing class hierarchy.
Keywords: Inheritance, Mixins, Multiple Inheritance, Traits, Reuse, Smalltalk

1 Introduction

Although single inheritance is widely accepted as the *sine qua non* of object-orientation, programmers have long realized that single inheritance is not expressive enough to factor out common features (*i.e.*, instance variables and methods) shared by classes in a complex hierarchy. As a consequence, language designers have proposed various forms of multiple inheritance [Mey88][Kee89][Str86], as well as other mechanisms, such as mixins [Moo86][BC90][FKF98], that allow classes to be composed incrementally from sets of features.

Despite the passage of nearly twenty years, neither multiple inheritance nor mixins have achieved wide acceptance [Tai96]. Summarizing Alan Snyder's contribution to the inheritance panel discussion at OOPSLA '87, Steve Cook wrote:

> "Multiple inheritance is good, but there is no good way to do it." [Coo87]

The trend seems to be away from multiple inheritance; the designers of recent languages such as Java and of C# decided that the complexities introduced by multiple inheritance outweighed its utility. It is widely accepted that multiple inheritance creates some serious implementation problems [DMVS89][SG99]; we believe that it also introduces serious *conceptual* problems. Our study of these problems has led us to the present design for traits.

* This research was partially supported by the National Science Foundation under award CCR-0098323.

Although multiple inheritance makes it possible to reuse any desired set of classes, a class is frequently not an appropriate element to reuse. This is because classes play two contradictory roles. A class has a primary role as a *generator of instances*: it must therefore be complete. But as a *unit of reuse*, a class should be small. These properties often conflict. Furthermore, the role of classes as instance generators requires that each class have a unique place in the class hierarchy, whereas units of reuse should be applicable at arbitrary places.

Moon's Flavors [Moo86] were an early attempt to address this problem: Flavors are small, not necessarily complete, and they can be "mixed in" at arbitrary places in the class hierarchy. More sophisticated notions of mixins were subsequently developed by Bracha and Cook [BC90], and Flatt, Krishnamurthi and Felleisen [FKF98]. These approaches all permit the programmer to create components that are designed for reuse, rather than for instantiation.

Mixins use the ordinary single inheritance operator to extend various base classes with the same set of features. However, although this inheritance operator is well-suited for deriving new classes from existing ones, it is not appropriate for composing reusable building blocks. Specifically, mixins must be composed linearly using inheritance; this severely restricts our ability to specify the glue code that is necessary to adapt the mixins so that they fit together.

In our proposal, lightweight entities called *traits* serve as the primitive units of code reuse. The design of *traits* started from the observation that the conflict between the goals of reuse and understandability is more apparent than real. In general, we believe that understanding a program is easier if it is possible to view the program in multiple forms. Even though a class may have been *constructed* by composing small traits in a complex hierarchy, there is no need to require that it be *viewed* in the same way. It should be possible to view the class *either* as a flat collection of methods *or* as a composite entity built from traits. The flattened view promotes understanding; the hierarchic view promotes reuse. There is no conflict so long as both of these views can coexist, which requires that the hierarchy is used only as a structuring tool and has *no effect on the meaning of the class*.

Traits satisfy this requirement. They provide structure, modularity and reusability *within* classes, but they can be ignored when one looks at the way that classes relate to each other. Traits provide an excellent balance between reusability and understandability, while enabling better conceptual modelling. Moreover, because traits are concerned solely with the reuse of behaviour, and not with the reuse of state, they avoid all of the implementation difficulties that characterize multiple inheritance and mixins. Traits have the following properties.

- A trait *provides* a set of methods that implement behaviour.
- A trait *requires* a set of methods that parameterize the provided behaviour.
- Traits do not specify any state variables, and the methods provided by traits never directly access state variables.
- Traits can be composed: trait composition is symmetric and conflicting methods are *excluded* from the composition.
- Traits can be nested, but the nesting has no semantics for classes—nested traits are equivalent to *flattened* traits.

A class can be constructed from a set of traits by inheriting from a superclass, and providing the necessary state variables and the required methods. These methods represent *glue* that specifies how the traits are connected together and how conflicts are resolved. This approach allows a class to be decomposed into a set of coherent features, and factors out the glue code that connects the features together. Because the semantics of a method is independent of whether it is defined in a trait or in a class that uses the trait, it is always possible to *flatten* a nested trait structure at any level.

The contributions of this paper are the precise identification of the reusability and understandability problems associated with multiple inheritance and mixins, and the presentation of traits as a composition model that solves these problems. We proceed as follows: in Section 2 we describe the problems of multiple inheritance and mixins, and in Section 3 we present traits and illustrate their use on some small examples. In Section 4 we summarize a formal model for traits. In Section 5, we discuss the most important design decisions and evaluate traits against the problems we identified in Section 2. In Section 6, we present our implementation of traits. In Section 7, we summarize the results of a realistic application of traits: a refactoring of the Smalltalk-80 collection hierarchy. We discuss related work in Section 8. We conclude the paper and indicate future work in Section 9.

2 Reusability Problems with Inheritance

Inheritance is commonly regarded as one of the fundamental features of object-oriented programming, but at the same time, inheritance is also a mechanism with many competing and often contradictory meanings and interpretations [Tai96]. Over the years, researchers have developed various inheritance models including single inheritance, multiple inheritance, and mixin inheritance. We give a brief overview of these models and point out their conceptual and practical problems regarding reusability. In particular we describe specific problems of mixin composition that have not been identified previously in the literature.

Note that this section is focused on reusability issues. Other problems with inheritance such as implementation difficulties [DMVS89][SG99] and conflicts between inheritance and subtyping [Ame90][MMMP90][LP91] are outside the scope of this paper.

Single Inheritance. Single inheritance is the simplest inheritance model; it allows a class to inherit from (at most) one superclass. Although this model is well-accepted, it is not expressive enough to allow the programmer to factor out all the common features shared by classes in a complex hierarchy. Hence single inheritance sometimes forces *code duplication*. Note that extension of single inheritance with interfaces as promoted by Java addresses the issues of subtyping and conceptual modeling, but does not provide any help with the problem of code duplication.

Multiple Inheritance. Multiple inheritance enables a class to inherit features from more than one parent class, thus providing the benefits of better code reuse and more

flexible modeling. However, multiple inheritance uses the notion of class in two contradictory roles, namely as the generator of instances and as the smallest unit of code reuse. This causes the following problems and limitations.

Conflicting features. One of the problems with multiple inheritance is the ambiguity that arises when conflicting features are inherited along different paths [DT01]. A particularly problematic situation is the "diamond problem" [BC90] (also known as "fork-join inheritance" [Sak89]) that occurs when a class inherits from the *same* base class via multiple paths. Since classes are instance generators, they need to provide some minimal behaviour (*e.g.*, methods `=`, `hash`, and `asString`), which is typically enforced by making them inherit from a common root class (*e.g.*, Object). However, this is precisely what causes the conflicts when several of these classes are reused.

Conflicting features manifest themselves as *conflicting methods* and *conflicting state variables*. Whereas method conflicts can be resolved relatively easily (*e.g.*, by overriding), conflicting state is more problematic. Even if the declarations are consistent, it is not clear whether conflicting state should be inherited once or multiply [Mey88][Sak92].

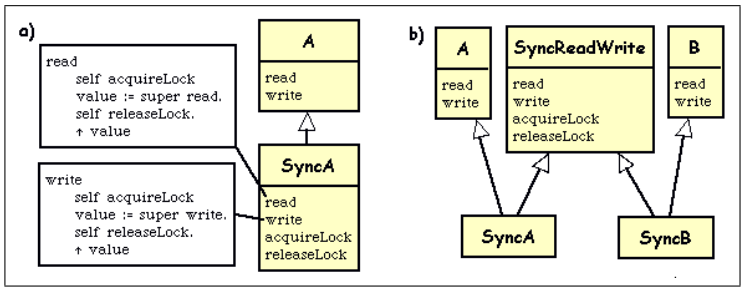

Fig. 1. In (a), the synchronization code is directly implemented in the subclass `SyncA`. In (b) we show an attempt to reuse the synchronization code for both `SyncA` and `SyncB`. This is impossible, because the methods in `SyncReadWrite` cannot refer to the `read` and `write` methods defined in `A` and `B`.

Accessing overridden features. Since identically named features can be inherited from different base classes, a single keyword (*e.g.*, `super`) is not enough to access inherited methods unambiguously. For example, C++ [Str86] forces one to explicitly name the superclass to access an overridden method; recent versions of Eiffel [Mey97] suggest the same technique[1]. This leads to tangled class references in the source code and makes the code vulnerable to changes in the architecture of the class hierarchy. Explicit superclass references are avoided in languages such

[1] The ability to access an overridden method using the keyword `Precursor` followed by an optional superclass name was added to Eiffel in 1997 [Mey97]. In earlier versions of Eiffel, access to original methods required repeated inheritance of the same class [Mey92]

as CLOS [Ste90] that impose a linear order on the superclasses. However, such a linearization often leads to unexpected behaviour [DH87][DHHM92] and violates encapsulation, because it may change the parent-child relationships among classes in the inheritance hierarchy [Sny86][Sny87].

Limited compositional power. Multiple inheritance allows a class to reuse features from multiple base classes. But unlike mixin inheritance, it does not allow one to write a reusable entity that both uses and exports adapted forms of methods implemented in unrelated classes[2].

This limitation is illustrated in Figure 1. Assume that class `A` contains methods `read` and `write` that provide unsynchronized access to some data. If it becomes necessary to synchronize access, we can create a class `SyncA` that inherits from `A` and overrides the methods `read` and `write` so that they call the inherited implementation under control of a lock (see Figure 1a).

Now suppose that class `A` is part of a framework that also contains another class `B` with `read` and `write` methods, and that we want to use the same technique to create a synchronized version of `B`. Naturally, we would like to factor out the synchronization code so that it can be reused in both `SyncA` and `SyncB`.

With multiple inheritance, the only way of sharing code among different classes is to inherit from a common superclass. This means that we have to move the synchronization code into a class `SyncReadWrite` that will become the superclass of both `SyncA` and `SyncB` (see Figure 1b). But a superclass cannot *explicitly* refer to a method like `read` that a possible subclass inherits from another superclass. It is possible to *implicitly* access such a method, by calling an abstract method on self that will eventually be implemented by the subclass. However, the whole point of this example is that unsynchronized reads *are not* and *should not* be available in `SyncA`! Thus, the class `SyncReadWrite` cannot access the `read` and `write` method provided by `A` and `B`, and it is not possible to factor out all the necessary synchronization code into `SyncReadWrite`.

Mixin Inheritance. A mixin is an abstract subclass specification that may be applied to various parent classes to extend them with the same set of features. Mixins allow the programmer to achieve better code reuse than single inheritance while maintaining the simplicity of the inheritance operation. However, although inheritance works well for extending a class with a single orthogonal mixin, it does not work so well for composing a class from many mixins. The problem is that usually the mixins do not *quite* fit together, *i.e.*, their features may conflict, and that inheritance is not expressive enough to resolve such conflicts. This problem manifests itself in various guises.

Total ordering. Mixin composition is linear: all the mixins used by a class must be inherited one at a time. Mixins appearing later in the order override *all* the identically named features of earlier mixins. Where conflicts should be resolved by selecting features from different mixins, a suitable total order may not exist.

[2] In C++ and Eiffel, parameterized structures such as templates [Str94] and generic classes [Mey92] compensate for this limitation.

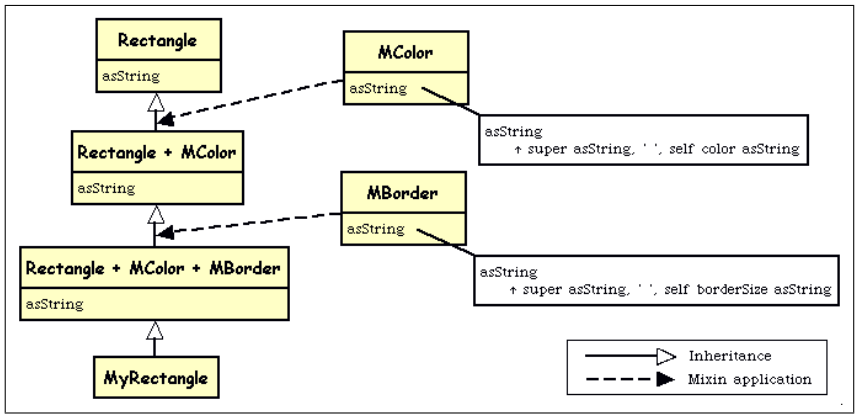

Fig. 2. The code that interconnects the mixins is specified in the mixin `MBorder`. The composite entity `MyRectangle` cannot access the implementations of `asString` in the mixin `MColor` and the class `Rectangle`. The classes containing `+` in their names are intermediate classes generated by mixin application.

Dispersal of glue code. With mixins, the composite entity is not in full control of the way that the mixins are composed: the conflict resolution code must be hardwired in the intermediate classes that are created as the mixins are used, one at a time. Obtaining the desired combination of features may require modifying the mixins, introducing new mixins, or, sometimes, using the same mixin twice.

As an example, consider the situation shown in Figure 2, where a class `MyRectangle` uses two mixins `MColor` and `MBorder` that both provide a method `asString`. The implementations of these methods in the mixins first call the original implementation via the keyword `super` and then extend the resulting string with specific information about their own state. When we compose the two mixins to make the class `MyRectangle`, we can choose which of them should come first, but we cannot specify how the two implementations should be composed. This is because the mixins must be added one at a time: in `Rectangle + MColor + MBorder` we can access the behaviour of `MBorder` and the *mixed* behaviour of `Rectangle + MColor`, but not the original behaviour of `MColor` and `Rectangle`.

Fragile hierarchies. Because of the strict linearity and the limited expressiveness regarding conflict resolution, composing multiple mixins results in inheritance chains that are fragile with respect to change. Adding a new method to one of the mixins may silently override an identically named method of a mixin that appears earlier in the chain. It may furthermore be impossible to reestablish the original behaviour of the composite without having to add or change several mixins in the inheritance chain. This is especially critical if one modifies a mixin that is used in many places across the class hierarchy.

As an illustration, suppose that in the previous example the mixin `MBorder` does not initially define a method `asString`. This means that the implementation of

`asString` in `MyRectangle` is the one specified by `MColor`. At a later point, suppose that the method `asString` is added to the mixin `MBorder`. Because of the total order, this implicitly overrides the implementation provided by `MColor`. Worse, the original behaviour of the composite class `MyRectangle` cannot be reestablished without changing more of the involved mixins.

3 Traits

We propose a compositional model as a solution to the problems illustrated in the previous section. Our model is based on lightweight entities called *traits*, which serve as the basic building blocks for classes and the primitive units of code reuse. Thus, traits satisfy the needs for structure, modularization and reusability *within* classes.

Traits, and all the examples given in this paper, are implemented in the Squeak dialect of Smalltalk-80 [IKM+97], but we believe that traits could also be applied to other single inheritance languages such as Java.

In this section we present the details of traits by using a running example. We show how classes are composed from traits, how traits are composed from other traits, and how naming conflicts are resolved.

3.1 Running Example and Notational Conventions

Suppose that we want to represent graphical objects such as circles or squares that can be drawn on a canvas. We will use traits to structure the classes and factor out the reusable behaviour. We focus on the representation of circles, but the same techniques can be applied to the other classes.

In the examples, trait names start with the letter T, and class names do not. We italicize required methods and embolden glue methods. Because the traits are implemented in Squeak, we present the code in Smalltalk. The notation `ClassName>>methodName` indicates that the method `methodName` is defined in the class `ClassName`.

3.2 Specifying Traits

A trait contains a set of methods that implement the behaviour that it *provides*. In general, a trait may *require* a set of methods that parameterize the provided behaviour. Traits cannot specify any state, and never access state directly. Trait methods can access state indirectly, using required methods that are ultimately satisfied by accessors (getter and setter methods).

The purpose of traits is to decompose classes into reusable building blocks by providing first-class representations for the different aspects of the behaviour of a class. Note that we use the term “aspect” to denote an independent, but not necessarily cross-cutting, concern. Traits differ from classes in that they do not define any kind of state, and that they can be composed using mechanisms other than inheritance.

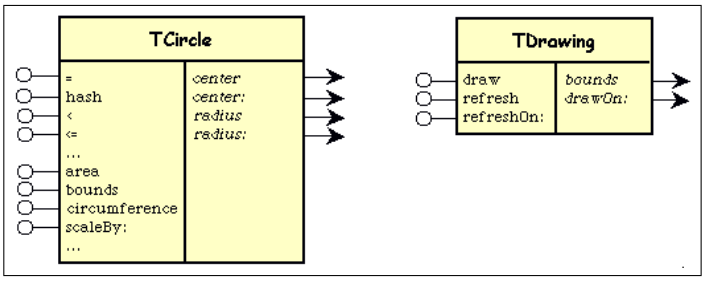

Fig. 3. The traits `TDrawing` and `TCircle` with provided methods in the left column and required methods in the right column.

Example. In our example, each graphical object can be decomposed into two aspects—its geometry, and the way that it is drawn on a canvas. In case of a circle, we represent the geometry with the trait `TCircle` and the behaviour necessary for drawing the object with the trait `TDrawing`.

Figure 3 shows these traits in an extension to UML. For each trait, the left column lists the provided methods and the right column lists the required methods. The trait `TDrawing` provides the methods `draw`, `refreshOn:`, and `refresh`, and it is parameterized by the required methods `bounds` and `drawOn:`. The code implementing this trait is shown below. The existence of the requirements is captured by methods (shown in italics) with body `self requirement`.

```
Trait named: #TDrawing uses: {}

draw                                          bounds
  ^self drawOn: World canvas                    self requirement

refresh                                       drawOn: aCanvas
  ^self refreshOn: World canvas                 self requirement

refreshOn: aCanvas
  aCanvas form
    deferUpdatesIn: self bounds
    while: [self drawOn: aCanvas]
```

The trait `TCircle` represents the geometry of a circle; it contains methods such as `area`, `bounds`, `circumference`, `scaleBy:`, `=`, `<`, and `<=`. `TCircle` requires methods `center`, `center:`, `radius`, and `radius`, which parameterize its behaviour. The implementation of this trait is shown in Appendix A.

3.3 Composing Classes from Traits

Traits are a completely downwards compatible extension of single inheritance. This means that trait composition does not subsume single inheritance; trait composition and inheritance are complementary. Whereas inheritance is used to derive one class from another, traits are used to achieve structure and reusability *within* a class definition. We

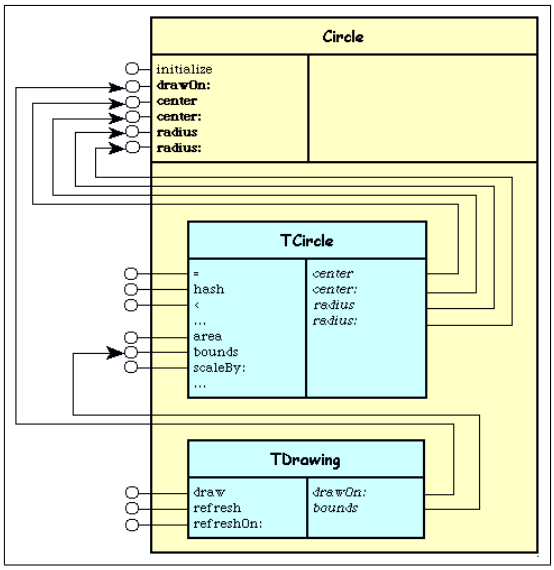

Fig. 4. The class `Circle` is composed from the traits `TCircle` and `TDrawing`. The requirement for `TDrawing>>bounds` is fulfilled by the trait `TCircle`. All the other requirements are fulfilled by accessor methods specified by the class.

summarize this relationship with the equation

$$Class = State + Traits + Glue$$

This means that a class is built by using a set of traits, adding the necessary state variables, and implementing the *glue methods* that connect the traits together and serve as accessors for the variables. In order for a class to be *complete*, all the requirements of the traits must be satisfied, *i.e.*, a method with the appropriate name must be provided. These methods can be implemented in the class itself, in a direct or indirect superclass, or by another trait that is used by the class.

Trait composition enjoys the *flattening property*, which says that the semantics of a method defined in a trait is identical to the semantics of the same method defined in a class that uses the trait. Specifically, this means that the keyword `super` has no special semantics for traits; it simply causes the method lookup to be started in the superclass of the class that *uses* the trait.

Another property of trait composition is that the composition order is irrelevant, and hence conflicting trait methods must be explicitly disambiguated (cf. Section 3.5). Conflicts between methods specified in classes and methods specified by incorporated traits are resolved using the following two precedence rules:

– *Class methods take precedence over trait methods.*
– *Trait methods take precedence over superclass methods.* This follows from the flattening property, which states that trait methods behave as if they were implemented in the class itself.

Example. As illustrated in Figure 4 and by the class definition hereafter, we create the class `Circle` by composing the traits `TCircle` and `TDrawing`. The trait `TDrawing` requires the methods `bounds` and `drawOn:`. The trait `TCircle` provides a method `bounds` which already fulfills one of the requirements. Therefore, the class `Circle` has to provide only the methods `center`, `center:`, `radius`, and `radius:` for the trait `TCircle` and the method `drawOn:` for the trait `TDrawing`.

The methods `center`, `center:`, `radius`, and `radius:` are simply accessors to two instance variables. The method `drawOn:` draws a circle on the canvas that is passed as the argument. In addition, the class also implements a method for initializing the two instance variables.

```
Object subclass: #Circle
    instanceVariableNames: 'center radius'
    traits: { TCirle . TDrawing }

initialize
  center := 0@0.
  radius := 50

center                                          center: aPoint
  ^center                                         center := aPoint

radius                                          radius: aNumber
  ^radius                                         radius := aNumber

drawOn: aCanvas
  aCanvas fillOval: self bounds color: Color black
```

3.4 Nested Traits

In the same way that classes are composed from traits, traits can be composed from other traits. Unlike classes, most traits are not complete, which means that they do not define all the methods that are required by their subtraits. Unsatisfied requirements of subtraits simply become required methods of the composite trait. Again, the composition order is not important, and methods defined in the composite trait take precedence over the methods of its subtraits.

Even in case of deeply nested traits, the flattening property remains valid. The semantics of a method does not depend on whether it is defined in a trait or in entities that are using this trait (cf. Section 5.1).

Example. The trait `TCircle` contains two different aspects: namely comparison operators and geometric functions. In order to separate these aspects and improve code reuse, we therefore redefine this trait as the composition of the traits `TMagnitude` and `TGeometry` as shown in Figure 5(a). Also the trait `TMagnitude` is specified as a nested trait; it uses the trait `TEquality`, which requires the methods `hash` and `=`, and provides the method $\sim$`=`. The trait `TMagnitude` itself requires `<`, and provides methods such as `max:`, `<=`, `between:and:`, and `>=`. Note that `TMagnitude` does not provide any of the methods required by its subtrait `TEquality`, which means that these requirements are just propagated as requirements of `TMagnitude`. Finally as shown below, theTrait `TCircle` is composed from the traits `TMagnitude` and `TGeometry`. It defines the required methods `=`, `hash`, and `<` for the trait `TMagnitude`.

In the following, we show only the definition of `TCircle`. The first line of this definition contains the *composition clause*, which specifies that `TCircle` uses the subtraits `TMagnitude` and `TGeomery`.

```
Trait named: #TCircle uses: { TMagnitude . TGeometry }

= other                                          hash
  ^self radius = other radius                      ^self radius hash
    and: [self center = other center]                and: [self center hash]

< other
  ^self radius < other radius
```

3.5 Conflict Resolution

A conflict arises if and only if we combine two traits providing identically named methods that do not originate from the same trait. Because traits cannot specify state, this explains why the diamond problem does not arise with traits; if the *same* method is obtained twice from different traits, there is no conflict (cf. Section 5.2). Based on the trait composition rules presented in Section 3.3, method conflicts must be explicitly resolved by defining a method in the class or in the composite trait. Traits enforce explicit resolution of conflicts by excluding the conflicting methods and therefore turning them into required methods.

To grant access to conflicting methods and thereby avoid code duplication, traits support the concept of *aliases*. Aliases allow one to make a trait method available under another name if the original name is excluded due to a conflict. Aliases are discussed further in Section 5.1.

In addition to conflict resolution, trait composition also supports *exclusion*, which allows one to avoid a conflict before it occurs. The composition clause allows a programmer to exclude methods from a trait when it is composed. This suppresses these methods and turns them into requirements, which can then be fulfilled by the otherwise conflicting implementations provided by other traits.

Example. To draw colored circles, a circle must contain color behaviour. To make this behaviour reusable, we specify it in the trait `TColor` shown in Figure 5(b). This trait provides the usual color methods such as `red`, `green`, `saturation`, *etc.*. Because colors can also be tested for equality, `TColor` uses the trait `TEquality`, and implements the required methods `=` and `hash` as shown below.

```
Trait named: #TColor uses: { TEquality }

hash                                        = other
  ^self rgb hash                              ^self rgb = other rgb
```

When the trait `TColor` is incorporated into the class `Circle`, a conflict arises because the traits `TColor` and `TCircle` provide different implementations for the methods `=` and `hash` as shown in Figure 5(c). Note that the method $\sim$`=` does not give rise to a conflict because in both `TCircle` and `TColor` the implementation originates from the same trait, namely `TEquality`.

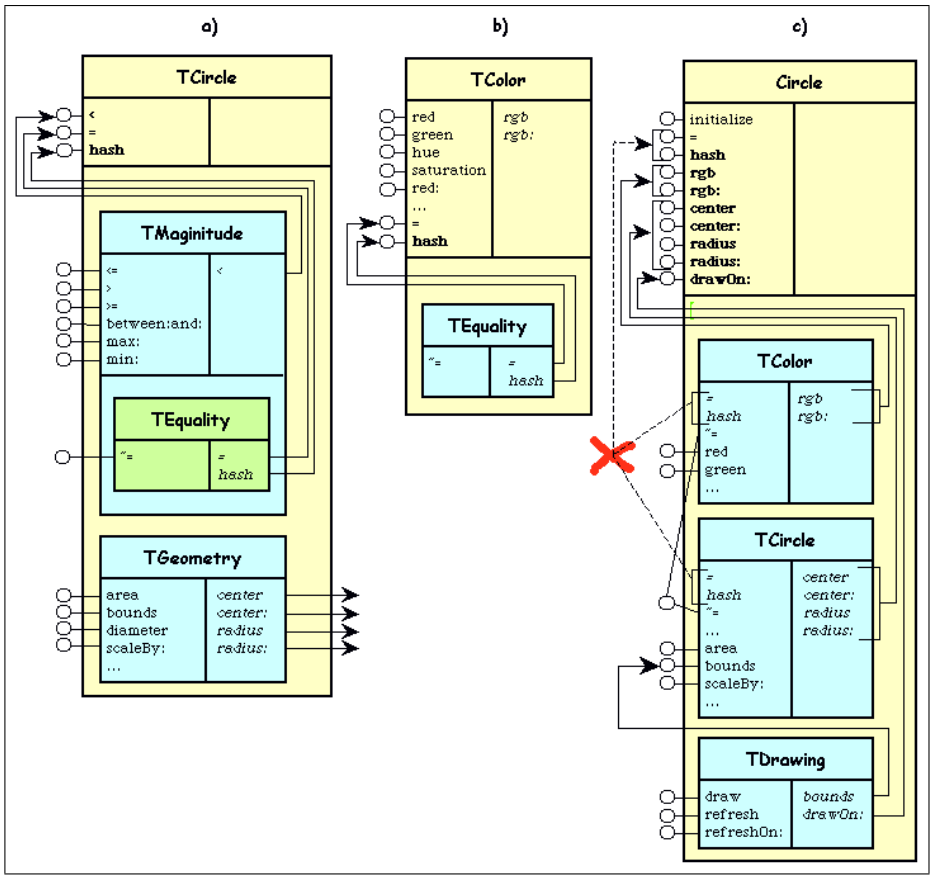

Fig. 5. Figure (a) shows how a trait `TCircle` is composed from a trait `TGeometry` and a nested trait `TMagnitude`, which is again composed from the trait `TEquality`. Note that the provided services of the nested traits are propagated to the composite trait (*e.g.*, `max:`, ∼=, and `area`), and similarly, the unsatisfied requirements of the nested traits (*e.g.*, `center` and `radius:`) are turned into required methods of the composite trait. In (b), we again use the trait `TEquality` to specify the comparison behaviour of the trait `TColor`. Figure (c) shows how a class `Circle` is specified by composing the traits `TCircle`, `TColor`, and `TDrawing`.

Figure 5(c) shows that the conflicting methods are excluded and thereby turned into requirements that have to be implemented in the class `TCircle` to make it complete. In the code shown below, we define the method `=` so that two colored circles are equal if and only if they have the same geometrical properties and the same color. To avoid code duplication, we specify aliases `circleEqual:`, `circleHash`, `colorEqual:`, and `colorHash` for the conflicting methods and use them to define the semantics of the composite.

```
Object subclass: #Circle
    instanceVariableNames: 'center radius rgb'
    traits:  { TCircle @ {#circleHash -> #hash. #circleEqual: -> #=} . TDrawing .
               TColor  @ {#colorHash -> #hash. #colorEqual: -> #=} }

hash                                        = anObject
  ^self circleHash                            ^(self circleEqual: anObject)
    bitXor: self colorHash                      and: [self colorEqual: anObject]
```

Alternatively, we might decide that equality of colored objects is independent of the color and only takes the geometrical properties into account. In this case, we could remove the conflicting methods `=`, `hash`, and $\sim$`=` from `TColor`. This avoids the conflicts and has the effect that the class `Circle` simply uses the comparison behaviour provided by the trait `TCircle`. The corresponding composition clause looks as follows:

```
Object subclass: #Circle
    instanceVariableNames: 'center radius rgb'
    traits:  { TCirle . TDrawing - {#=. #hash. #~=} . TColor }
```

4 Formal Traits Model

Space does not allow us to present the formal model of traits in these proceedings. Instead we summarize the model briefly; interested readers will find full details in a technical report [SDNB02]. This summary glosses over many details, including fields, metaclasses, and the lifting of the requires and provides functions from methods to classes.

The model abstracts away from the details of any particular language, and assumes only the existence of methods and sets of definitions; each definition binds a name to a method. We also assume a base language with single inheritance; if C is a class and D a set of definitions, then the class D `extends` C is a subclass of C in the usual way.

There are five ways of constructing a trait T in the model.

$T ::= D$		(a simple set of definitions)
	D `with` T	(some definitions D using a nested trait T)
|	$T_1 + T_2$	(the symmetric composition of two traits)
|	$T - x$	(the trait T excluding the definition for name x)
|	$T[x \rightarrow y]$	(the trait T with the addition of an alias x)

The semantics of a trait is represented as a record, *i.e.*, a finite mapping from names to methods. Finding the record corresponding to each of these syntactic structures is straightforward. D `with` T is defined to mean the record corresponding to T overridden by the record corresponding to D. $T_1 + T_2$ means a record containing all of the names

that are defined only in T_1 and all of the names that are defined only in T_2. Name clashes, *i.e.*, names that are defined in both T_1 and T_2, annihilate each other and do not appear in the resulting record.

The aliasing operation $T[x \rightarrow y]$ is defined to mean the record corresponding to T with the addition of a new mapping from x to whatever y maps to in T. If x is already defined in T, then the two definitions annihilate each other, and the resulting record has no mapping at x.

The model then defines *structured classes* S as a superset of ordinary unstructured classes. In addition to the ordinary inheritance operation D `extends` S, a structured class D `with` T `extends` S can be built by combining some local definitions D, a trait T and a (possibly structured) superclass S.

The meaning of inheritance is given by representing each class (structured or unstructured) as a sequence of records. This sequence is an abstraction of the superclass chain; the first record of the sequence models the methods defined in the class itself, the next record models the methods defined in its superclass, and so on up to the top of the inheritance chain. The reason that we keep the whole sequence is to capture the semantics of `super` ; although the exact meaning of `super` may vary from one language to another, we assume that if we model the whole inheritance chain, we will be able to capture the appropriate semantics.

Given this semantic domain, the meaning of D `extends` C is given by prepending the record corresponding to D to the meaning of C; the same is true for D `extends` S. The meaning of D `with` T `extends` S is also given by prepending a *single* record to the meaning of S. The record that is prepended is the record corresponding to T overridden by the new definitions in D. The important point to note is that defining a structured class D `with` T `extends` S adds only one element to the sequence of records for S. This captures the fact that occurrences of `super` in D and occurrences in T both refer to methods in S, and that `super` in D never refers to a method in T.

A consequence of this construction is that the sequence of records that represents the semantics of a structured class S also represents the semantics of some unstructured class C; we call C the *flattening* of S. This allows the programmer to work with a class in both the flattened and the structured forms. The model also allows us to prove that $+$ is associative and commutative, *i.e.*, that trait composition is unordered.

5 Discussion and Evaluation

In this section, we discuss some design decision that significantly influenced the properties of traits. We focus on reusability and understandability of programs that are written using traits. Finally, we present an evaluation of traits against the reusability problems discussed in Section 2.

5.1 Design Decisions

Traits were designed with other reusability models in mind: we tried to combine their advantages, while avoiding their disadvantages. Here, we discuss the most important design decisions.

Untangling Reusability and Classes. Although they are inspired by mixins, traits constitute a new kind of conceptual entity that represents a finer-grained unit of reuse than a class while not being tied to a specific place in the inheritance hierarchy. We believe that this is essential for improving code reuse and conceptual modelling in object-oriented programming languages for two reasons. First, traits close the conceptual gulf that lies between entire classes and single methods; it allows classes to be built by composing reusable behaviours rather than by implementing a large and unstructured set of methods. Second, it separates two contradictory roles of classes: instance generators and reusable method repositories [BHJL86]. As instance generators, classes are typically organized in hierarchies in order to make their instances *complete*, but units of reuse should be arbitrarily *small* and their reusability should not be subject to hierarchy restrictions.

Single Inheritance and the Flattening Property. Instead of replacing single inheritance, we decided to keep this familiar concept and simply extend it with the concept of trait composition. These two concepts are similar but complementary and work together nicely.

Single inheritance allows one to reuse all the features (*i.e.*, methods and state variables) that are available in a class. If a class can only inherit from a single superclass, inheriting state does not cause complications and a simple keyword (*e.g.*, `super`) is enough to access overridden methods. This form of accessing inherited features is very convenient, but it also assigns semantics to the place of a method in the inheritance hierarchy.

Trait composition operates at a finer granularity than inheritance; it is used to modularize the behaviour defined *within* a class. As such, trait composition is designed to compose only behaviour and not state. In addition, trait composition enjoys the flattening property, which means that it does not assign any semantics to the place where a method is defined.

The flattening property in combination with single inheritance demonstrates that traits are a logical evolution of the single inheritance paradigm. A system based on traits not only allows one to write and execute traditional single inheritance code, but even if there are thousands of deeply nested traits, with appropriate tool support, the user can still *view and edit* the classes in *exactly* the same way as if the system were implemented without using traits at all.

Aliasing. Many multiple inheritance implementations allow one to access overridden features by explicitly naming the respective superclass in the source code. In C++, this is done with the scope operator `::` , whereas Eiffel uses the keyword `Precursor` followed by the superclass name enclosed in curly brackets. With traits, we chose method aliasing over named trait references in method bodies to avoid the following problems.

– Named trait references contradict the *flattening property*, because they prevent the creation of a semantically consistent flattened view without adapting these references in the method bodies.

– Named trait references would require the trait structure to be hardcoded in all the methods that use this construct. This means that changing the trait structure or simply moving methods from one trait to another potentially invalidates many methods.
– Named trait references would require an extension of the syntax of the underlying single inheritance language.

Method aliasing avoids all of these problems. Specifically, aliasing conforms to the flattening property because the flattening process can simply introduce a new name for the aliased method body.

Although the concept of method aliasing has some similarities to method renaming as provided by Eiffel, there are essential differences. Whereas aliasing just establishes an alternative name without affecting the original one, with renaming the original method name becomes undefined. As a consequence, method renaming must change all the references to the old name so that they refer to the new one, whereas aliasing has no effect on any references. Changing the bodies of methods in this way would violate the flattening property.

5.2 Evaluation Against the Identified Problems

In Section 2 we identified a set of conceptual and practical reusability problems that are associated with various forms of inheritance. The design of traits was significantly influenced by the attempt to solve these problems. In the following, we present a point by point evaluation of the results.

Conflicting features. Since trait composition supports composing several traits in parallel, conflicting features are also an issue. However, the problem is much less serious with traits. Traits cannot define state, so the *diamond problem* does not arise. Although a class may obtain the same method from the same trait via multiple paths, these multiple copies do not give rise to a conflict, and will therefore be unified.

Accessing overridden features. With traits, we decided against approaches based on naming the superclass in the source code of the methods (as used by Eiffel and C++) or on linearization (as used by CLOS). Instead, we decided to use a simple form of method aliasing as described in Section 5.1. This avoids both tangled class references in the source and code that is hard to understand and fragile with respect to changes.

Limited compositional power. Like mixins, traits can *explicitly* refer to a method implemented by the superclass of the class that uses the trait. So the problem illustrated in Figure 1 can be solved by implementing the synchronization methods `read`, `write`, `acquireLock`, and `releaseLock` in a reusable trait. This trait is then used in both `SyncA` and `SyncB`, which do not need to implement any methods other than accessors for the lock variable.

Total ordering. Trait composition does not impose total ordering, but it can express ordering by means of nesting. In addition, trait composition can be combined with inheritance, which allows a wide variety of partially ordered compositions.

Dispersal of glue code. When traits are combined, the glue code is always located in the combining entity, reflecting the idea that the superordinate entity is in complete charge of plugging together the components that implement its aspects. This property nicely separates the glue code from the code that implements the different aspects. This makes a class easy to understand, even if it is composed from many different components.

Fragile hierarchies. Since traits are designed to be used in many different classes, robustness with respect to change has been a leading principle in designing trait composition. In particular, traits require every method conflict to be resolved explicitly. The consequence is that resolving conflicts requires some extra work, but that the behaviour of the composite is what the programmer expects.

In addition, any problem caused by changes to a trait is limited to the direct user of that trait, whether that be a class or a composite trait. This is because the user is always in complete control of how the components are plugged together. With mixins this is not so, as was discussed in section 2: introducing a single new method into a mixin may require the programmer to change *many* other components, or to introduce new components, at each place where the mixin is used. With traits, change is *localized*: a single change in a component requires at most one compensating change in each direct user of the component in order to reestablish the original behaviour.

6 Implementation

Traits as described in this paper are implemented in Squeak [IKM+97], an open-source Smalltalk-80 dialect. Our implementation consists of two parts: an extension of the Smalltalk-80 language and an extension of the integrated development environment (IDE).

6.1 Language Extension

To introduce traits, we extended the implementation of a class so that it includes an additional field to contain the information in the composition clause. This field defines the traits used by the class, including any exclusions and aliases. In addition, we introduced a representation for traits, which are essentially stripped down classes that can define neither state nor a superclass. When a class `C` uses a trait `T`, the method dictionary of `C` is extended with an entry for all the methods in `T` that are not overridden by `C`. For an alias, we simply extend the method dictionary with an entry that associates the alternative name with the aliased method. Since the compiled methods in traits do not usually depend on the location where they are used, the bytecode for the method can be shared between the trait that defines the method and all the classes and traits that use it. However, methods using the keyword `super` store an explicit reference to the superclass in their literal table. So we need to copy those methods and change the entry for the superclass appropriately. Copying could be avoided by modifying the virtual machine to compute `super` when needed.

In Smalltalk, classes are first-class objects; every class is instance of a metaclass that defines the shape and the behaviour of its singleton instance [GR83]. In our implementation, we support this concept by introducing the notion of a *metatrait*; a metatrait can be associated with every trait. When a trait is used in a class, the associated metatrait (if there is one) is automatically used in the metaclass. Note that a trait without a metatrait can be applied to both classes and metaclasses. To preserve metaclass compatibility [Gra89][BSLR98], metatraits are automatically generated for traits that send methods to the metalevel using the pseudo-message `class` .

Because traits are simple and completely downward compatible with single inheritance, implementing traits in a reflective single inheritance language like Squeak is unproblematic. The fact that traits cannot specify state is a major simplification. We avoid most of the performance and space problems that occur with multiple inheritance, because these problems are related to compiling methods without knowing the indices of the slots in the final layout of the object [DMVS89]. In fact, our implementation requires no duplication of source code, and byte code is duplicated only if it includes sends to `super` . A program with traits shows essentially the same performance as a corresponding single inheritance program where all the methods provided by traits are implemented directly in the classes using the traits. This is especially remarkable because our implementation did not require any changes to the Squeak virtual machine. There may be a small performance penalty resulting from the use of accessor methods, but such methods are in any case widely used because they improve maintainability.

6.2 Programming Tools

Besides an extension of the language, our implementation also includes an extension of the programming tools, *i.e.*, the Smalltalk browser. For each class (and each trait), the browser shows the different traits from which it is composed. The flattening property allows the browser to flatten this hierarchical structure at any nesting level. In addition, the browser shows the programmer the *provided* and *required* methods, the *overridden* methods, and the *glue* methods, which specify how the class meets the requirements of its component traits. These features help the programmer to work with different views of the code. On the one hand, the programmer can work with the code in a flattened view, where a class consists of an unstructured set of methods and it does not matter whether the class is built from traits and whether a method is defined in a trait or in the class itself. On the other hand, the programmer can work in a composition view, where he sees how the responsibilities of the class are decomposed into several traits and how these traits are glued together in order to achieve the required behaviour. This view is especially valuable because it allows a user to understand a class by knowing the involved traits and understanding the glue methods.

As in standard Smalltalk, the browser supports incremental compilation. Whenever a trait method is added, changed or excluded, all the users of that trait are instantaneously updated. The modifications are also analyzed to infer the set of required methods. If a modification causes a new conflict or an unspecified requirement anywhere in the system, the affected classes and traits are automatically added to a “to do” list.

Our implementation features several tools that support the programmer in composing traits and generating the necessary glue code. Required methods, for example,

can automatically be mapped to instance variables by generating the necessary accessor methods. Conflict resolution is also semi-automated by presenting the programmer with a list of alternative implementations; choosing one of these automatically generates the composition clause that excludes the others, and thus eliminates the conflict.

7 A Realistic Application of Traits

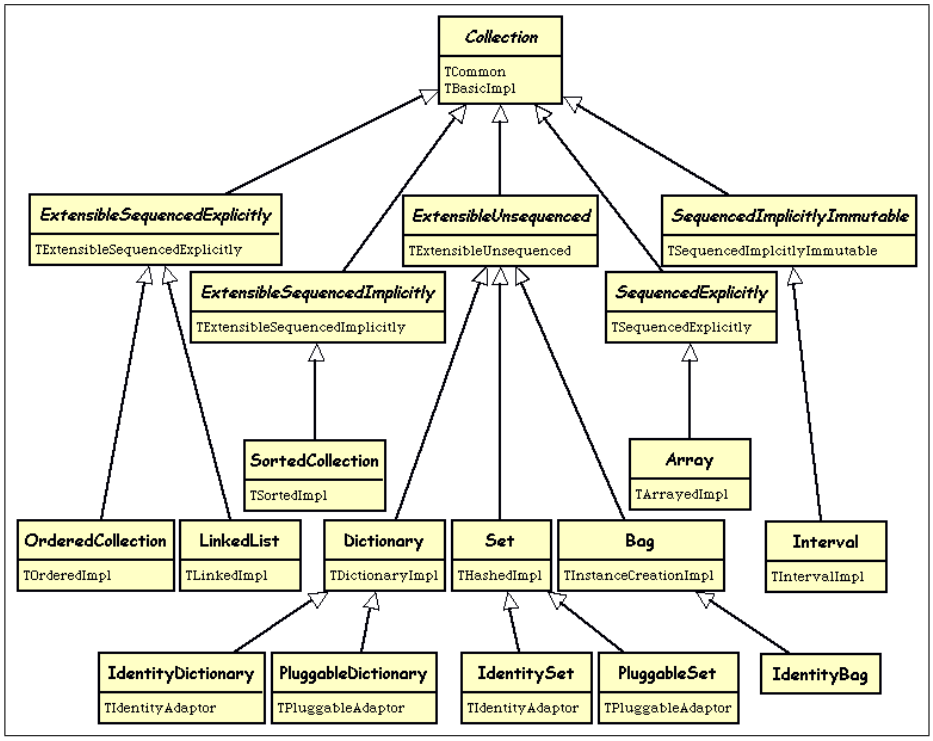

Fig. 6. This shows the refactored collection hierarchy. Classes with italicized names are abstract; below the class name we show the traits that are used by the class directly.

As a realistic evaluation of their usability, we used traits to refactor the Smalltalk-80 collection hierarchy as it is implemented in Squeak 3.2. In this Section, we summarize the results of this work; interested readers are referred to a companion paper that contains more details [BSD02].

The core classes of the Smalltalk-80 collection hierarchy have been improved over more than 20 years and are often considered a paradigmatic example of object-oriented programming. Each kind of collection can be characterized by properties such as explicitly ordered (*e.g.*, Array), implicitly ordered (*e.g.*, `SortedCollection`), unordered (*e.g.*, `Set`), extensible (*e.g.*, `Bag`), immutable (*e.g.*, `String`), keyed (*e.g.*,

`Dictionary`), element comparison (*e.g.*, using identity or a higher-level comparison operator), *etc*.

The problem is that single inheritance is not expressive enough to model such a diverse set of related classes that share many different properties in various combinations. This means that the implementors of the hierarchies are forced to duplicate code or to move methods higher in the hierarchy and then disable them in the subclasses to which they do not apply [Coo92].

Traits allowed us to solve these problems by creating traits for the different collection properties and combining them to build the required collection classes. In order to achieve maximum flexibility, we separated the properties specifying the implementation of a collection from the properties specifying the interface. This allowed us to freely combine different interfaces (*e.g.*, "sorted-extensible interface" and "sorted-extensible-immutable interface") with any of the suitable implementations (*e.g.*, "linked-list implementation" and "array-based implementation").

In addition to the traits that were absolutely necessary in order to achieve a sound hierarchy and avoid code duplication, we structured the code in more fine-grained subtraits that allows us to reuse parts of the code outside of the collection hierarchy. As an example, we introduced traits representing the behaviour "emptiness" (which requires `size` and provides `isEmpty`, `notEmpty`, `ifEmpty:`, *etc.*) and "enumeration" (requires `do:` and provides `collect:`, `select:`, `detect:`, *etc.*).

Although some of the collection classes are now built as the nested composition of up to 20 traits, the flattening property combined with the corresponding programming tools means that this does not impact understandability: it is always possible to work with the hierarchy as if it were implemented with ordinary single-inheritance.

Figure 6 shows the refactored hierarchy for 13 of the more common collection classes. Besides the class name, it also shows the traits that the class uses. However, it does not show that each of these traits has up to 20 subtraits. At the top, there is the abstract class `Collection`, which provides a small amount of general behaviour for all collections. Then we have a layer of abstract classes that provide different combinations of traits representing interface properties. At the bottom, we have concrete classes that use traits to provide implementations.

In total, these classes use 46 different traits and implement 509 methods, whereof 36 are automatically generated accessor methods. This is just over 5% fewer methods than in the original implementation. In addition, the code for the trait implementation is 12% smaller than the original. This is especially remarkable because 10% of the methods in the original implementation are implemented "too high" in the hierarchy specifically to enable code sharing. With inheritance, the penalty for this is the repeated need to cancel inherited behaviour (using methods that cause a runtime error) in subclasses where they do not make sense. In the trait implementation, there is no need to resort to this tactic.

8 Related Work

In the Section 2 we have already shown how other inheritance schema try to promote code reuse. Therefore in this section we compare traits only to some existing models that we did not consider previously.

There are at least two other models that use entities called “traits” as an approach to share and reuse implementation. One of them is the prototype based language SELF [US87]. In SELF, there is no notion of class; each object conceptually defines its own format, behaviour (methods), and inheritance relations. Objects are derived from other objects by cloning and then modifying them. In addition, SELF also has the notion of *traits objects* that serve as repositories for sharing behaviour and state among multiple objects. Traits are used as dictionaries during method lookup and there is no mechanism for resolving conflicts.

The software for the Xerox Star workstation also used entities called *traits* as an approach to multiple inheritance [CBLL82]. This approach has more in common with other multiple inheritance approaches than with the trait model presented in this paper. Some of the main differences from our model are that the Star traits have a different semantics regarding inheritance, have different conflict resolution capabilities, carry state, and allow multiple implementations for a single method.

9 Conclusions and Future Work

This paper introduces traits, a simple compositional model for building and structuring object-oriented programs. Traits are composed using a set of operators—symmetric combination, exclusion, and aliasing—that are carefully designed so that they allow a fair amount of composition flexibility without being subject to the problems and limitations that we have identified for mixins and multiple inheritance.

Thanks to the favorable composition properties, traits are an ideal extension for single inheritance languages. Traits are completely downwards compatible and do not require modifying or extending the method syntax of the underlying language. Furthermore, the flattening property guarantees optimal understandability of the resulting code, because it is always possible to both view and edit the code as if it were written using single-inheritance.

Having the right programming tools has proven to be crucial for giving the programmer the maximum benefit from traits. In our Squeak-based implementation, we changed the browser so that it allows the programmer to switch seamlessly between the different views and emphasizes the glue methods that define how the traits are connected.

We successfully used traits for refactoring the collection hierarchy, which is a strong indication for the usability of traits for realistic and non-trivial problems. It also showed that traits are suitable for modularizing classes that are already built, and that they raise the level of abstraction when building new classes. Finally, working with the refactored hierarchy impressed us with the power of the flattening property for understanding classes that are built from multiple and deeply nested traits.

As future work we would like to (1) evaluate the impact of the introduction of visibility mechanisms on the flattening property, (2) refine the calculus that captures the formal model, (3) evaluate the possibility of using traits modify the behaviour of single instances at run-time, (4) develop a type systems for traits and identify the relationships between traits and interfaces, and (5) further explore the application of traits to refactoring of complex class hierarchies.

Acknowledgements. We would like to thank Gilad Bracha, William Cook, Erik Ernst, Robert Hirschfeld, Andreas Raab, and Roel Wuyts for their rich interaction and valuable comments while developing traits and writing this paper.

References

[Ame90] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In *Proc. REX/FOOLS Workshop*, Noordwijkerhout, June 1990. to appear.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices*, pages 303–311, October 1990. Published as Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices, volume 25, number 10.

[BHJL86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the emerald system. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, pages 78–86, November 1986. Published as Proceedings OOPSLA '86, ACM SIGPLAN Notices, volume 21, number 11.

[BSD02] Andrew Black, Nathanael Schärli, and Stéphane Ducasse. Applying traits to the smalltalk collection hierarchy. Technical Report CSE-02-014, OGI School of Science & Engineering, Oregon Health & Science University, 2002.

[BSLR98] Noury M. N. Bouraqadi-Saadani, Thomas Ledoux, and Fred Rivard. Safe metaclass programming. In *OOPSLA 1998 Proceedings*, pages 84–96, 1998.

[CBLL82] Gael Curry, Larry Baer, Daniel Lipkie, and Bruce Lee. TRAITS: an approach to multiple inheritance subclassing. In *Proceedings ACM SIGOA, SIGOA Newsletter*, Philadelphia, June 1982. Published as Proceedings ACM SIGOA, SIGOA Newsletter, volume 3, number 12.

[Coo87] Steve Cook. Oopsla '87 panel p2: Varieties of inheritance. In *OOPSLA '87 Addendum To The Proceedings*, pages 35–40. ACM Press, October 1987.

[Coo92] William R. Cook. Interfaces and specifications for the smalltalk-80 collection classes. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 1–15, October 1992. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.

[DH87] R. Ducournau and Michel Habib. On some algorithms for multiple inheritance in object-oriented programming. In J. Bézivin, J-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings ECOOP'87*, volume 276 of *LNCS*, pages 243–252, Paris, France, June 15-17 1987. Springer-Verlag.

[DHHM92] R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. Monotonic conflict resolution mechanisms for inheritance. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 16–24, October 1992. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.

[DMVS89] R. Dixon, T. McKee, M. Vaughan, and Paul Schweizer. A fast method dispatcher for compiled languages with multiple inheritance. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, pages 211–214, October 1989. Published as Proceedings OOPSLA '89, ACM SIGPLAN Notices, volume 24, number 10.

[DT01] Dominic Duggan and Ching-Ching Techaubol. Modular mixin-based inheritance for application frameworks. In *Proceedings OOPSLA 2001, ACM SIGPLAN Notices*, pages 223–240, October 2001.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–183. ACM Press, 1998.

[GR83] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.

[Gra89] Nicolas Graube. Metaclass compatibility. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, pages 305–316, October 1989. Published as Proceedings OOPSLA '89, ACM SIGPLAN Notices, volume 24, number 10.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pages 318–326, November 1997.

[Kee89] Sonia E. Keene. *Object-Oriented Programming in Common-Lisp*. Addison Wesley, 1989.

[LP91] W. LaLonde and John Pugh. Subclassing = subtyping = is-a. *Journal of Object-Oriented Programming*, 3(5):57–62, January 1991.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[MMMP90] Ole Lehrmann Madsen, Boris Magnusson, and Birger Moller-Pedersen. Strong typing of object-oriented languages revisited. In *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices*, pages 140–150, October 1990. Published as Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices, volume 25, number 10.

[Moo86] David A. Moon. Object-oriented programming with flavors. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, pages 1–8, November 1986. Published as Proceedings OOPSLA '86, ACM SIGPLAN Notices, volume 21, number 11.

[Sak89] Markku Sakkinen. Disciplined inheritance. In S. Cook, editor, *Proceedings ECOOP'89*, pages 39–56, Nottingham, July 10-14 1989. Cambridge University Press.

[Sak92] Markku Sakkinen. The darker side of C++ revisited. *Structured Programming*, 1992. to appear.

[SDNB02] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: The formal model. Technical Report CSE-02-013, OGI School of Science & Engineering, Oregon Health & Science University, 2002.

[SG99] Peter F. Sweeney and Joseph (Yossi) Gil. Space and time-efficient memory layout for multiple inheritance. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 256–275. ACM Press, 1999.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, pages 38–45, November 1986. Published as Proceedings OOPSLA '86, ACM SIGPLAN Notices, volume 21, number 11.

[Sny87] Alan Snyder. Inheritance and the development of encapsulated software systems. In *Research Directions in Object-Oriented Programming*, pages 165–188. MIT Press, 1987.

[Ste90] Guy L. Steele. *Common Lisp The Language*. Digital Press, second edition, 1990. book.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, Reading, Mass., 1986.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison Wesley, 1994.

[Tai96] Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, September 1996.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, December 1987.

A TCircle

The implementation of the trait `TCircle` as discussed in Section 3.2.

```
Trait named: #TCircle uses: {}

= other
 ^self radius = other radius
   and: [self center = other center]

hash
 ^self radius hash and: [self center hash]

< other
 ^self radius < other radius

scaleBy: factor
 self radius: factor * self radius

max: other
 ^self > other
   ifTrue: [self]
   ifFalse: [other]

between: min and: max
 ^self >= min and: [self <= max]

bounds
 ^Rectangle
   origin: self center - self radius
   corner: self center + self radius

center
  self requirement

radius
  self requirement
```

```
~= other
  ^(self = other) not

<= other
  ^(self > othet) not

> other
  ^other < self

min: other
  ^self < other
     ifTrue: [self]
     ifFalse: [other]

area
  ^self radius * Float pi squared

circumference
  ^2 * self radius * Float pi

diameter
  ^2 * self radius

>= other
  ^(self < other) not

center: aPoint
  self requirement

radius: aNumber
  self requirement
```