Traits: A Mechanism for Fine-grained Reuse

Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, Andrew Black

▶ To cite this version:

Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, Andrew Black. Traits: A Mechanism for Fine-grained Reuse. ACM Transactions on Programming Languages and Systems (TOPLAS), 2010, 28. ⟨hal-05025979⟩

HAL Id: hal-05025979
https://hal.science/hal-05025979v1

Submitted on 8 Apr 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Traits: A Mechanism for Fine-grained Reuse

STÉPHANE DUCASSE
Software Composition Group, University of Berne and
Language and Software Evolution Group, LTI — LISTIC, Université de Savoie
and
OSCAR NIERSTRASZ and NATHANAEL SCHÄRLI
Software Composition Group, University of Berne
and
ROEL WUYTS
Lab for Software Composition and Decomposition, Université Libre de Bruxelles
and
ANDREW P. BLACK
Department of Computer Science, Portland State University

Inheritance is well-known and accepted as a mechanism for reuse in object-oriented languages. Unfortunately, due to the coarse granularity of inheritance, it may be difficult to decompose an application into an optimal class hierarchy that maximizes software reuse. Existing schemes based on single inheritance, multiple inheritance, or mixins, all pose numerous problems for reuse. To overcome these problems we propose *traits*, pure units of reuse consisting only of methods. We develop a formal model of traits that establishes how traits can be composed, either to form other traits, or to form classes. We also outline an experimental validation in which we apply traits to refactor a non-trivial application into composable units.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and objects, inheritance*; D.2.7 [**Software Engineering**]: Distribution and Maintenance—*Restructuring*

General Terms: Languages

Additional Key Words and Phrases: Inheritance, Mixins, Multiple Inheritance, Traits, Reuse, Smalltalk

1. INTRODUCTION

Inheritance in object-oriented languages is well-established as an incremental modification mechanism that can be highly effective at enabling code reuse between similar classes [Wegner and Zdonik 1988]. Unfortunately, single inheritance is inadequate for expressing classes that share features not inherited from their (unique) common parent. The shared features must either be forced into the common parent (where they do not belong), or they

©2006 Stéphane Ducasse. Author's address: Software Composition Group, University of Berne, Neubrückstrasse 10, CH-3012 Berne, Switzerland and LISTIC - ESIA, 5, chemin de Bellevue, Domaine universitaire d'Annecy-Le-Vieux, France.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

must be duplicated in the classes that should share them. To overcome this limitation, language designers have proposed various forms of multiple inheritance [Meyer 1988; Keene 1989; Stroustrup 1986], as well as other mechanisms, such as mixins [Moon 1986; Bracha and Cook 1990; Mens and van Limberghen 1996; Flatt et al. 1998; Ancona et al. 2000], 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 [Taivalsaari 1996]. 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." [Cook 1987]

Not only does multiple inheritance pose serious implementation problems [Dixon et al. 1989; Sweeney and Gil 1999], it is often inappropriate as a reuse mechanism: although multiple inheritance makes it possible to reuse a class (or a set of classes), a class is frequently not the element that one wishes to reuse. This is because classes play two competing roles. A class is primarilly a *generator of instances*. Therefore, most of the recent object-oriented programming languages such as Java and C# make every class bundle together a *complete* set of basic features by requiring it to be a (direct or indirect) subclass of the dedicated class Object. A class has a secondary role as a *unit of reuse*. It should therefore bundle a *minimal* set of features which can sensibly be reused together[1]. Unfortunately these two roles conflict. Since classes must adopt a fixed position in the class hierarchy (i) it can be difficult or impossible to factor out wrapper methods (*i.e.*, methods that extend other methods with additional functionality) as reusable classes, (ii) conflicting features inherited from different paths may be difficult to resolve, and (iii) overridden features may be difficult to access or compose. Perhaps for these reasons the designers of recent languages such as Java and C# decided that the complexities introduced by multiple inheritance outweigh its utility.

Flavors [Cannon 1982; Moon 1986] were an early attempt to address these problems: Flavors are small, incomplete implementations of classes, that can be "mixed in" at arbitrary places in the class hierarchy. More sophisticated notions of mixins were subsequently developed by Bracha and Cook [Bracha and Cook 1990], Mens and van Limberghen [Mens and van Limberghen 1996], Flatt, Krishnamurthi and Felleisen [Flatt et al. 1998], and Ancona, Lagorio and Zucca [Ancona et al. 2000].

Mixins use the ordinary single inheritance operator to extend various parent classes with the same set of features. Although this inheritance operator is well-suited for deriving new classes from existing ones, it is not necessarily appropriate for composing reusable building blocks. Specifically, because mixin composition is implemented using inheritance, mixins are composed linearly. This gives rise to several problems. First, a suitable total ordering of features may be difficult to find, or may not even exist. Second, "glue code" that exploits or adapts the linear composition may be dispersed throughout the class hierarchy. Third, the resulting class hierarchies are often fragile with respect to change, so that conceptually simple changes may impact many parts of the hierarchy. For these reasons, we believe, mixins have never achieved wide success in mainstream object-oriented languages.

Traits represent a simple solution to these various dilemmas. In a nutshell, a trait is *a set of methods*, divorced from any class hierarchy. Traits can be composed in arbitrary order.

[1] Note that this paper focusses on code reuse, and hence will not discuss the relation between classes and types, nor interfaces.

The composite entity has complete control over the composition and can resolve conflicts explicitly, without resorting to linearization. Classes are organized in a single inheritance hierarchy, and can make use of traits purely to specify the incremental difference in behavior with respect to their superclasses. This simple model has the following consequences.

—Two roles are clearly separated: traits are purely units of reuse, and classes are generators of instances.

—Traits are simple software components that both *provide* and *require* methods (*required methods* are those that are used by, but not implemented in, a trait).

—Classes are composed from traits, in the process resolving any conflicts, and possibly providing the required methods.

—Traits specify no state, so the only conflict that can arise when combining traits is a method conflict. Such a conflict can be resolved by overriding or by exclusion.

—Traits can be inlined, a process that we call "flattening": the fact that a method originates in a trait rather than in a class does not affect the semantics of the class.

—Difficulties experienced with multiple inheritance disappear with traits, because traits are divorced from the inheritance hierarchy.

—Difficulties experienced with mixins also disappear, because traits impose no composition order.

This paper extends our earlier work [Schärli et al. 2003] by presenting a precise and formal account of traits, a more detailed analysis of the problems associated with the different multiple inheritance and mixin mechanisms, and a more extensive discussion of related work. More information about traits can be found in Schärli's dissertation [Schärli 2005].

In this paper we present a formal model of traits and demonstrate how traits resolve numerous problems with existing approaches to specifying classes. In Section 2, we give an overview of the problems arising with multiple inheritance and mixins, and in Section 3 we show in some detail how these problems affect existing programming languages. In Section 4 we present traits, and illustrate their use by means of numerous examples. Section 5 discusses the most important design decisions and evaluates traits with respect to the problems identified in Section 2, while Section 6 presents our implementation of traits. In Section 7 we summarize the results of realistic applications of traits: a refactoring of the Smalltalk collection hierarchy, a refactoring of the Smalltalk language kernel by bootstrapping it with traits, and an application of traits to address the problem of safe metaclass composition. We discuss related work in Section 8. We conclude the paper and indicate future work in Section 9.

2. PROBLEMS WITH INHERITANCE AND COMPOSABILITY

Although inheritance is widely considered to be the key feature of object-oriented programming, it is also saddled with many competing and contradictory definitions and interpretations [Taivalsaari 1996]. Over the years, researchers have developed various forms of inheritance, including single inheritance, multiple inheritance, and mixin inheritance. Each of these forms provides different answers to problems of *decomposition* — how we decompose a software base into suitable units of reuse — and *composition* — how we compose these units to obtain a class hierarchy suitable for our application domain.

In this section, we give an overview of the key problems of decomposition and composition and how they apply to the different forms of inheritance. For conciseness, this

overview is kept general; details of the inheritance mechanisms in particular programming languages are deferred to Section 3. Note that we focus here on conceptual issues related to reuse. Other problems with inheritance such as implementation difficulties [Dixon et al. 1989; Sweeney and Gil 1999] and conflicts between inheritance and subtyping [America and van der Linden 1990; Bruce et al. 1995; Castagna 1995; Cook et al. 1990; Madsen et al. 1990; LaLonde and Pugh 1991] are outside the scope of this paper.

2.1 Decomposition Problems

Object-oriented programming offers a tool for modeling arbitrary domains as hierarchies of classes. But the way in which we decompose our domain concepts into classes is not necessarily the right way to decompose the implementations of these classes into sets of features [LaLonde 1989; Harrison and Ossher 1993; Tarr et al. 1999]. Let us briefly consider three decomposition problems in this context.

Duplicated Features. Single inheritance is the simplest form of inheritance; it allows a class to inherit from (at most) one superclass [Cook and Palsberg 1989]. Although well-accepted, single inheritance is not expressive enough to factor out all the common features shared by classes in a complex hierarchy. As a consequence, single inheritance sometimes forces code to be duplicated.

As an example, consider the Smalltalk stream classes ReadStream, WriteStream, and ReadWriteStream. As suggested by their names, the class ReadWriteStream contains features provided by both ReadStream and WriteStream. However, single inheritance allows ReadWriteStream to inherit from only one of these classes. In Smalltalk, ReadWriteStream inherits from WriteStream and then duplicates some methods from ReadStream.

Note that extension of single inheritance with interfaces as promoted by Java and C# addresses the issues of subtyping and conceptual modeling, but does not provide any help with the problem of duplicated code.

Inappropriate Hierarchies. A common way of avoiding such code duplication is to implement certain methods “too high” in the hierarchy. The idea is that instead of duplicating a method, it is moved to a superclass until it is available in all the classes where it is actually required. In our example, this means that the programmer could implement all the reading methods in the class PositionableStream, which is the lowest common superclass of ReadStream and WriteStream. As a consequence, these methods would be inherited in the class ReadWriteStream, and hence would not need to be duplicated.

The tactic succeeds, but the price is high: PositionableStream is polluted by many methods that have nothing to do with positioning, and WriteStream appears to implement many reading methods, although these methods will fail or result in inconsistent behavior if they are ever used.

Both multiple inheritance and mixins attempt to alleviate these problems by allowing a class to obtain features from multiple sources, but, as we shall see, each gives rise to other problems.

Duplicated Wrappers. Multiple inheritance as provided by languages such as C++ and Eiffel enables a class to reuse features from multiple parent classes, but it does not allow one to write a reusable entity that extends methods implemented in as-yet unknown classes.

This limitation is illustrated in UML class diagrams in Figure 1. Assume that class A contains methods read and write: that provide unsynchronized access to some data. (If not

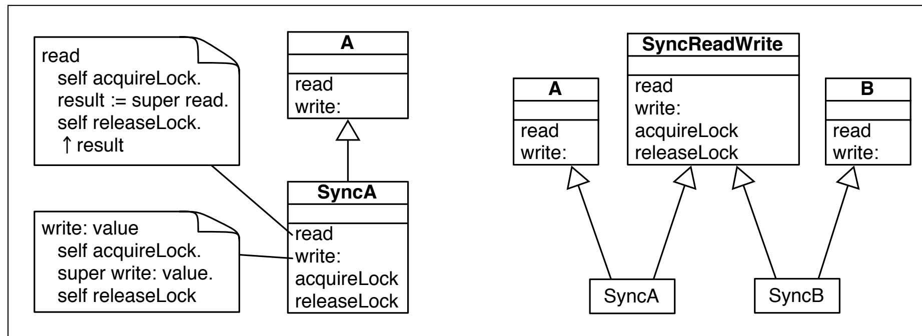


Fig. 1. Incorrect attempt to factor out a synchronization wrapper

otherwise indicated, all the code shown is written in Smalltalk. The traits model, however, is not specific to Smalltalk.) If it becomes necessary to synchronize access, we can create a class SyncA that inherits from A and wraps the methods read and write:. That is, SyncA defines new read and write: methods that call the inherited methods under control of a lock (see Figure 1 left).

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 natural way to share code among different classes is to inherit from a common superclass. This means that we should move the synchronization wrapper into a class SyncReadWrite that will become the superclass of both SyncA and SyncB (see Figure 1 right). Unfortunately this does not work because super-sends are statically resolved in most forms of multiple inheritance such as those of C++ and Eiffel. Therefore, the super-sends in methods of SyncReadWrite would refer *statically* to methods of its superclass, and not to methods in A or B.

Workarounds are clumsy and just entail more duplicated code, for example, the super-calls in SyncReadWrite could be replaced by calls to abstract methods directRead and directWrite:, which are then implemented in both SyncA and SyncB to call, respectively, the read and write: methods of A and B. (See Section 3.2 for more details.)

Mixins solve this particular problem by late-binding super. A mixin is an abstract subclass specification that may be applied to various parent classes to extend them with the same set of features [Moon 1986; Bracha and Cook 1990; Mens and van Limberghen 1996; Flatt et al. 1998]. Instead of defining SyncReadWrite as a class, it is defined as a mixin. Then SyncA and SyncB will each apply the mixin to a different superclass, and obtain the desired wrapper behavior.

2.2 Composition Problems

Although there is a clear progression in expressive power from single inheritance through multiple inheritance to mixins, this expressiveness does not come without a cost. Both multiple inheritance and mixins pose numerous problems when we consider how classes are composed from shared features.

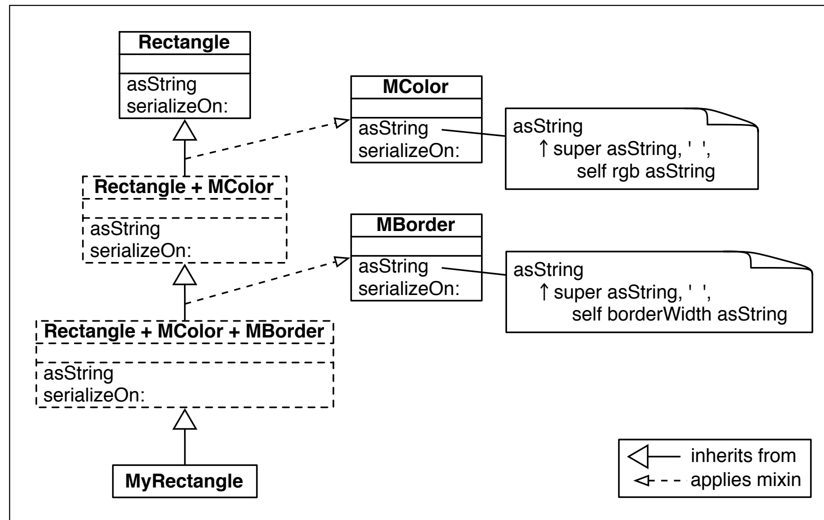


Fig. 2. Lack of composition control in the composite class MyRectangle

Conflicting Features. One of the problems with multiple inheritance is the ambiguity that arises when conflicting features are inherited along different paths [Duggan and Techaubol 2001]. A particularly problematic situation is the "diamond problem" [Bracha and Cook 1990] (also called "fork-join inheritance" [Sakkinen 1989]), which occurs when a class inherits from the same parent class via multiple paths. The root of a class hierarchy (or a subhierarchy) often consists of a class that provides some common default behavior that may be overridden by subclasses (*e.g.*, methods =, hash, and asString): this is precisely the cause of the conflicts that arise when several of these classes are reused.

The features that conflict may be methods or attributes (instance variables). Whereas method conflicts can be resolved relatively easily (*e.g.*, by overriding), conflicting attributes are more problematic. Even if the declarations are consistent, it is not clear whether conflicting attributes should be inherited once or multiply [Meyer 1988; Sakkinen 1992], and how these attributes should be initialized.

Single inheritance does not suffer from this problem; nor do mixins that are based on single inheritance. With mixin composition, mixins are applied to classes *one at a time*, generating new subclasses in a single inheritance hierarchy. Conflicts do not arise because the features of each mixin simply extend or override those of the class to which it is applied. However, the fact that mixins must be applied in a particular order leads to other problems, as we shall see shortly.

Lack of Control and Dispersal of Glue Code. Mixin composition is linear: all the mixins used by a class must be inherited one at a time. Features defined in mixins appearing later in the order override *all* the identically named features of earlier mixins. Where conflicts should be resolved by selecting and combining features from different mixins, a suitable total order may not exist. So, while avoiding the problem of ambiguous conflicts, mixins put feature composition into a straitjacket from which it may be difficult to escape.

As a consequence, with mixins, the composite entity is not in full control of the way in which the mixins are composed: instead, the way in which the individual features override

and extend one another is dictated by the total ordering imposed on the mixins. Obtaining the desired combination of features may require introducing glue code in new intermediate mixins, or even modifying the component mixins. Neither alternative is satisfactory. Modifying a mixin is problematic because it may break other classes that use the mixin; introducing intermediate mixins causes the glue code to be scattered throughout the inheritance hierarchy, which makes the composition hard to understand and adapt.

As an example, consider the class diagram shown in Figure 2, where a class MyRectangle uses two mixins MColor and MBorder that each provide methods asString and serializeOn:. (We introduce an *ad-hoc* extension to UML to show where mixins are applied in the inheritance hierarchy, and we use class names such as Rectangle + MColor for the anonymous intermediate classes that result when mixins are applied.) The implementations of the method asString in the mixins first call the implementation inherited from the superclass (using the keyword super) and then extend the resulting string with a separation character followed by some specific information about their own state. Similarly (but not shown in Figure 2), the implementations of serializeOn: in the mixins first call the superclass method and then append the mixin's own state to the argument stream.

Suppose now that for compatibility reasons, we need to serialize our class MyRectangle so that the rgb value appears before the borderWidth. Because mixin composition is linear, this means that the mixin MColor must be applied before the mixin MBorder. Unfortunately, this means that the order of the asString methods is also changed, so the color attributes will be printed before the border attributes, which may not be what we want.

The crux of the problem is that from within the composite entity MyRectangle, it is not possible to control separately how the different features are composed. This is because in MyRectangle, we can only access the *mixed* behavior of Rectangle + MColor + MBorder, but not the original behavior of MColor and Rectangle.

Thus, if we need to customize how the features are composed — be it because we need a different serialization order or a another separation character between the two strings — we need to modify the involved mixins, which is problematic as it potentially breaks all the other clients of these mixins. (See Section 3.1 for more details.)

Note that *composite mixins* [Bracha 1992] do not provide any more control over composition than do ordinary mixins. Like mixins, composite mixins provide only a linear composition in which all the features of the involved mixins are totally ordered. This means that, in the above example, the same problem would occur if we were to combine the two mixins MColor and MBorder to create a composite mixin MColorAndBorder and then use this composite mixin to define our new class MyRectangle. The only difference is that the problem would manifest itself during the definition of the composite mixin rather than the definition of the class MyRectangle.

Fragile Hierarchies. The total ordering of mixins can lead to a further problem, namely that the resulting inheritance hierarchies will often be fragile with respect to change. Adding a new method to one of the mixins *implicitly* overrides all identically named methods of mixins that appear earlier in the chain. It may furthermore be impossible to reestablish the original behavior of the composite without adding or changing 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 MyRect-

angle will be the one specified by MColor. Suppose that, later, the method asString is added to the mixin MBorder. Due to the total ordering of mixins, this implicitly overrides the implementation provided by MColor. Worse, the original behavior of the composite class MyRectangle cannot be reestablished without changing more of the mixins involved in the composition. If we want to avoid the ripple effect caused by further changes to existing mixins, we have to introduce a new "glue mixin" between the mixins MColor and MBorder, which makes the method asString provided by MColor available under a new name such as colorAsString, and then add another glue method asString to the class MyRectangle.

With many forms of multiple inheritance we also observe a fragility problem with respect to changes. Since identically named features can be inherited from different parent classes, a single keyword (*e.g.*, super) is not enough to access inherited methods unambiguously. For example, C++ [Stroustrup 1997] forces one to name the appropriate superclass in order to access an overridden method; recent versions of Eiffel [Meyer 1997] adopt an analogous technique[2]. Although explicitly naming the source of overridden features makes it possible to compose features from multiple parent classes, this embedding of explicit class references into the source code makes the code fragile with respect to changes in the class hierarchy.

3. DETAILED DISCUSSION OF THE PROBLEMS

The previous section provided a general outline of the problems associated with inheritance as a reuse mechanism. In this section we provide some detailed illustrations of how these problems apply to the various forms of inheritance adopted by some existing programming languages.

3.1 Mixins in Strongtalk and Jam

Strongtalk [Bak et al. 2002] and Jam [Ancona et al. 2000] are extensions of Smalltalk and Java with mixins. Both suffer from the limitations caused by the total ordering imposed by mixin composition. As an illustration, consider the situation shown in Figure 2, and suppose again that we need to serialize MyRectangle objects so that the rgb value appears on the stream before the borderWidth; this means that the mixin MColor has to be applied before the mixin MBorder.

As pointed out earlier, this also means that the color attributes will be printed before the border attributes; worse, it is not possible to change this ordering within the composite class MyRectangle. Reversing the printing order without duplicating any code would require us to modify the component mixins.

The need to modify the mixins arises because the asString methods include not only the mixin-specific printing behavior but also determine the composition order of the mixin code and the super code. This gives us the clue that separating these two concerns might allow for more flexible reuse. The first step is to remove from the mixin methods the code referring to the superclass implementation. In the Smalltalk-based language Strongtalk, the mixin code would look like this:

[2]The ability to access an overridden method using the keyword Precursor followed by an optional superclass name was added to Eiffel in 1997 [Meyer 1997]. In earlier versions of Eiffel, access to original methods required *direct repeated inheritance* [Meyer 1992]. This means that a subclass inherits from the same superclass (at least) twice so that one copy of a method can be redefined while the other still refers to the original version in the superclass.

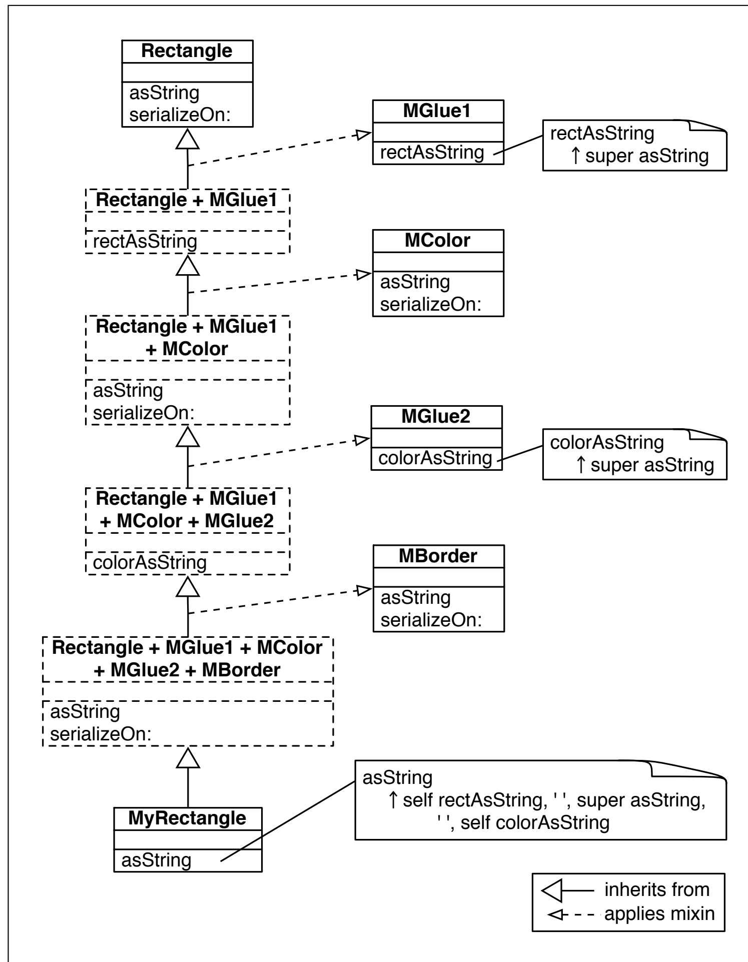


Fig. 3. Customizing the composition using two intermediate “glue mixins”

```
MColor>>asString³
    ↑ self color asString

MBorder>>asString
    ↑ self borderWidth asString
```

The second step is to make *all* of the asString methods accessible from the composite class MyRectangle; because the mixins are composed linearly, unless we do something extraordinary, the method asString provided by the superclass Rectangle is overridden by the method from MColor, and the method from MColor is overridden by the method from MBorder, so from within the composite class MyRectangle it is only the MBorder method that is accessible.

We make these methods accessible under additional names in the following way. We interpose two additional “glue mixins” MGlue1 and MGlue2 in the hierarchy, as shown in Figure 3. MGlue1 is inserted between the class Rectangle and the mixin MColor, and it

implements the method

```
MGlue1>>rectAsString
    ↑ super asString
```

to make the method Rectangle>>asString available under the name rectAsString. Similarly, MGlue2 is inserted between the mixins MColor and MBorder, and it implements the method MGlue2>>colorAsString to make the method MColor>>asString available under the name colorAsString. Now it is possible to write the glue method asString in the combining class MyRectangle in a way that gives us the printing order that we want:

```
MyRectangle>>asString
    ↑ self rectAsString, ' ', super asString, ' ', self colorAsString.
```

The problem with this approach is that interposing the two additional glue mixins makes the inheritance hierarchy more complex and causes the glue code to be scattered over three different entities. This makes program comprehension much more difficult. Even a programmer who is familiar with the class Rectangle and the mixins MColor and MBorder will have to look at three different entities to understand the composition of the mixins. Furthermore, in order to understand the behavior of the composite class MyRectangle, it is necessary to understand how six entities in the inheritance hierarchy are composed, and in particular how the super-sends of the mixins MGlue1 and MGlue2 are resolved.

Note that the even though the examples are presented in Strongtalk syntax, JAM suffers from exactly the same problems as it facilitates the same kind of mixin composition.

3.2 Multiple Inheritance and Mixins in C++

The language C++ [Stroustrup 1997] is quite unique from our perspective: it features native support for multiple inheritance, and can also be made to support mixins by using templates to represent classes with a parameterized superclass.

Multiple Inheritance. A distinctive feature of multiple inheritance in C++ is that the programmer has a certain amount of control over a diamond situation. If a base class (that is, a superclass) is declared to be *virtual*, the base class is shared and attributes are inherited only once[4]. While this provides help for avoiding conflicts and ambiguities in a diamond situation, it does not help us to solve the problem of factoring out generic wrappers.

Let us reconsider the example shown in Figure 1, where a class A implements two methods for reading and writing and has a subclass SyncA that implements synchronized versions of these methods. Figure 4 shows the implementation of SyncA in C++.

With multiple inheritance, sharing code among different classes means (directly or indirectly) inheriting from a common superclass that contains the code to be shared. Therefore, if we want to share the synchronization code in SyncA to create another synchronized subclass SyncB of B, we need to factor this code into a new class SyncReadWrite and then make it the superclass of both SyncA and SyncB (see Figure 1 right).

Unfortunately, multiple inheritance alone is not expressive enough to do this. The problem is that the calls to the superclass versions of read and write are statically bound and can refer only to a *superclass* of SyncReadWrite. Therefore, the class SyncReadWrite cannot

[4] In his description of C++ [Stroustrup 1997], Stroustrup uses the term “mixin” for a class that overrides methods of a virtual base class. This definition of “mixin” differs from that used in this paper and in most of the research literature.

```
class SyncA : public A {
    public:
    virtual int read() {
        acquireLock();
        result = A::read();
        releaseLock();
        return result;
    };
    virtual void write(int n) {
        acquireLock();
        A::write(n);
        releaseLock();
    };

    protected:
    virtual void acquireLock() {
        // acquire lock
    };
    virtual void releaseLock() {
        // release lock
    };
};
```

Fig. 4. The class SyncA in C++

```
class SyncReadWrite {
    public:
    virtual int read() {
        acquireLock();
        result = directRead();
        releaseLock();
        return result;
    };
    virtual void write(int n) {
        acquireLock();
        directWrite(n);
        releaseLock();
    };

    protected:
    virtual void acquireLock() {
        // acquire lock
    };
    virtual void releaseLock() {
        // release lock
    };

    virtual int directRead() = 0;
    virtual void directWrite(int n) = 0;
};
```

Fig. 5. The class SyncReadWrite implemented with two abstract methods

```
class SyncA : public A, SyncReadWrite {
    public:
    virtual int read() {
        return SyncReadWrite::read();
    };
    virtual void write(int n) {
        SyncReadWrite::write(n);
    };

    protected:
    virtual int directRead() {
        return A::read();
    };
    virtual void directWrite(n) {
        A::write(n);
    };
};
```

```
class SyncB : public B, SyncReadWrite {
    public:
    virtual int read()
        return SyncReadWrite::read();
    };
    virtual void write(int n) {
        SyncReadWrite::write(n);
    };

    protected:
    virtual int directRead() {
        return B::read();
    };
    virtual int directWrite(n) {
        B::write(n);
    };
};
```

Fig. 6. Code duplication in the classes SyncA and SyncB

```
template <class Super>
class MSyncReadWrite : public Super {
    public:
    virtual int read() {
        acquireLock();
        result = Super::read();
        releaseLock();
        return result;
    };
    virtual void write(int n) {
        acquireLock();
        Super::Write(n);
        releaseLock();
    };

    protected:
    virtual void acquireLock() {
        // acquire lock
    };
    virtual void releaseLock() {
        // release lock
    };
};

class SyncA : public MSyncReadWrite<A> {};

class SyncB : public MSyncReadWrite<B> {};
```

Fig. 7. Synchronization expressed as a mixin

```
template <class Super>
class MLogOpenClose : public Super {
    public:
    virtual void open() {
        Super::open();
        log("Opened");
    };
    virtual void close() {
        Super::close();
        log("Closed");
    };
    virtual void reset() {
        // reset logger
    };

    protected:
    virtual void log(char* s) {
        // write to log
    };
};

class MyDocument : public MSyncReadWrite<MLogOpenClose<Document>> {};
```

Fig. 8. The class MyDocument built from two mixins

explicitly call the unsynchronized versions of the methods read and write provided by its *subclasses* A and B.

As a workaround, one would have to modify the methods read and write in SyncReadWrite so that the explicit calls to the superclass methods are replaced by calls to abstract methods directRead and directWrite (Figure 5), which will then be implemented by the subclasses SyncA and SyncB (Figure 6). This solution is still far from satisfactory, since it requires duplication of four glue methods in each subclass. Furthermore, avoiding name clashes between the synchronized and unsynchronized versions of the read and write methods makes this approach rather clumsy, and one has to make sure that the unsynchronized methods directRead are not publicly available in SyncA and SyncB.

Template-based Mixins. Unlike the generics mechanisms of most other languages such as Java and C#, the C++ template mechanism allows the programmer to write classes with generic superclasses. As shown by VanHilst and Notkin [VanHilst and Notkin 1996a; 1996b] as well as Smaragdakis and Batory [Smaragdakis and Batory 1998; 2000], this enables the programmer to express a mixin as a class with a generic superclass. Thus, the C++ programmer can avoid the limitation of multiple inheritance with regard to wrappers by using mixins instead. In the previous example, this means that the synchronization code can be written as a generic class MSyncReadWrite. This generic class can then be used to create the classes SyncA and SyncB by applying it to the superclasses A and B, respectively. The corresponding code is shown in Figure 7.

Apart from the fact that C++ mixins are explicitly written as generic classes, this approach is identical to ordinary mixins as discussed earlier. Therefore, it is not surprising that it solves our problem without any code duplication, but also suffers from the linearization problems pointed out previously as soon as multiple mixins are composed.

As an example, assume that we want to combine the mixin MSyncReadWrite with another wrapper mixin MLogOpenClose to create a new class MyDocument, which differs from its superclass Document in that it synchronizes all the calls to the methods read and write, and logs all the calls to the methods open and close. Unfortunately, this again requires the programmer to choose an order for the two mixins. In the code shown in Figure 8, we decided to apply the mixin MSyncReadWrite last, which means that it overrides all the features of the other mixin MLogOpenClose. This is not a problem as long as the two mixins do not implement conflicting features. But it does make the whole hierarchy fragile with respect to changes: if the mixin MSyncReadWrite is changed so that it also provides a method reset, then this new method will *implicitly* override the implementation provided by MLogOpenClose and hence break our class MyDocument.

As illustrated in Section 3.1, the programmer can fix such conflicts by modifying existing mixins, which is problematic if the changed mixins are used elsewhere, or by introducing new intermediate mixins, which leads to dispersal of glue code. In addition, C++ offers a third option for resolving such conflicts. This option is based on the fact that C++ allows a class to explicitly access features of its indirect superclasses by using nested scope qualifiers.

In the previous example, this means that the class MyDocument can use the expression

```
MSyncReadWrite::MLogOpenClose::reset()
```

to refer to the method reset in MLogOpenClose, while it can use the expression

```
MSyncReadWrite::MLogOpenClose::Document::open()
```

to refer to the method open in Document.

Using such nested scope qualifiers, the programmer can write the glue code directly in the composite class MyDocument. This avoids the dispersal of glue code caused by introducing intermediate mixins, but it introduces other problems. Besides the fact that nested scope qualifiers make the code hard to read, understand and maintain, they also make the code fragile with respect to changes in the hierarchy and can break encapsulation [Snyder 1986]. This is because a class using such code not only depends on the entirety of features inherited from its direct superclass, but can also have explicit dependencies on the complete inheritance hierarchy (*e.g.*, the exact order of the applied mixins) and the implementation details of all its *indirect* superclasses (*e.g.*, whether a superclass implements or inherits a certain feature).

3.3 CLOS

Unlike C++ and Eiffel, the multiple inheritance variation of CLOS imposes a linear order on the superclasses. This has the advantage that a single keyword call-next-method is enough to unambiguously call a superclass method. As a consequence, CLOS avoids the fragility that is caused by allowing the programmer to include explicit superclass references in the source code of arbitrary methods. Another feature of CLOS is that super-sends are dynamically resolved, which means that CLOS can express and apply generic wrappers without any code duplication. Finally, CLOS doesn't have any problems with conflicting slots (fields) because they are treated essentially in the same way as methods. So if a class inherits multiple definitions for the same slot, the most specific definition takes precedence over the less specific ones, and their slot options are combined in useful ways.

On the downside, CLOS linearization [Steele 1990] leads to problems similar to the ones we identified and described for mixins. In particular, it often leads to unexpected behavior

because it is not always clear how a complex multiple inheritance hierarchy should be linearized [Ducournau et al. 1992]. Other linearizations used in Lisp and its derivatives (such as the schemes used in Loops [Stefik and Bobrow 1985], Dylan [Barrett et al. 1996] and C3 [Barrett et al. 1996]) do not provide fundamental solutions to these problems, since they still favor the automatic resolution of conflicts.

4. TRAITS — COMPOSABLE UNITS OF BEHAVIOR

Traits offer a simple solution to the problems outlined in Section 2. With this approach, classes retain their primary role as generators of instances, while traits are purely units of reuse. Classes are organized in a single inheritance hierarchy, thus avoiding the key problems of multiple inheritance, but the incremental extensions that classes introduce to their superclasses are specified using one or more traits.

In this section we introduce traits by means of a formal model, informal diagrams, and running examples. Traits are implemented in Squeak, a Smalltalk dialect [Ingalls et al. 1997], and have been applied successfully to refactor significant portions of the Squeak library [Black et al. 2003] and the Squeak language kernel [Ducasse et al. 2005]. As of now, traits are a standard feature of the statically typed language Scala [Odersky et al. 2004], and there is an implementation of Traits for Perl 5. Furthermore, there is ongoing work on porting traits to other languages such as C# (see Section 9) and making traits a standard feature of the upcoming Perl 6 (under the name "roles").

A trait is essentially a set of methods, *i.e.*, a mapping from method names to method bodies. Composite traits may be specified by means of compositions. In the formal model, these are expressions over traits using *trait sum* ($+$), *exclusion* ($-$) and *aliasing* ($\rightarrow$) operators. When traits are composed with $+$, identical names that map to different method bodies will conflict; we represent this by mapping the method name to $\top$ in the composite. The *overriding* operation $\triangleright$ is used to override these conflicts with proper method bodies.

Traits bear a superficial resemblance to mixins, with several important differences. Several traits can be applied to a class in a single operation, whereas mixins must be applied incrementally. Trait composition is unordered, thus avoiding problems due to linearization of mixins. Traits contain only methods, so state conflicts are avoided, but method conflicts may exist. A class is specified by composing a superclass with a set of traits and some *glue methods*. Glue methods are defined in the class and they connect the traits together; *i.e.*, they implement required trait methods (possibly by accessing state), they adapt provided trait methods, and they resolve method conflicts.

Trait composition respects the following three rules:

—*Methods defined in a class itself take precedence over methods provided by a trait.* This allows glue methods defined in the class to override methods with the same name provided by the traits.

—*Flattening property.* A non-overridden method in a trait has the same semantics as if it were implemented directly in the class.

—*Composition order is irrelevant.* All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated.

A *conflict* arises if we combine two or more traits that provide identically named methods that do not originate from the same trait. Conflicts are resolved by implementing a glue method at the level of the class, which overrides the conflicting methods, or by excluding a method from all but one trait. In addition traits allow *method aliasing*; this makes it

possible for the programmer to introduce an additional name for a method provided by a trait. The new name is used to obtain access to a method that would otherwise be unreachable because it has been overridden. As we shall see, method aliasing offers a less fragile solution to this problem than method renaming.

We shall first introduce our formal model by summarizing those aspects of classes that we need to capture. We will then proceed to define traits, and show how traits are used to build classes. We will use running examples to illustrate various aspects of the formal model.

4.1 Classes and Methods

A key feature of traits is that, although classes may be built using traits, the way in which this is done does not affect the semantics of classes. In effect, traits can be "inlined", or flattened. For this reason, we start by describing a model of classes without traits.

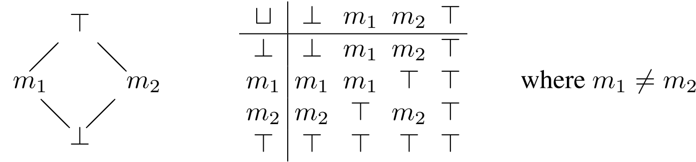
The primitive elements of our model are the following disjoint sets:

—$\mathcal{N}$, a countable set of method *names*, and

—$\mathcal{B}$, a countable set of method *bodies*,

—$\mathcal{A}$, a countable set of *attribute* names (*i.e.*, instance variables).

To express conflicts, we extend the set of method bodies $\mathcal{B}$ to a flat lattice $\mathcal{B}^\star$, with new elements $\bot$ and $\top$ such that $\bot \sqsubset m \sqsubset \top$, for all $m \in \mathcal{B}$, and in which all other elements are incomparable. We will use $\bot$ to represent a required (undefined) method and $\top$ to represent a method conflict. Thus, the *least upper bound* or *join* operator $\sqcup$ for $\mathcal{B}^\star$ is as follows:



$\sqcup$	$\bot$	m_1	m_2	$\top$
$\bot$	$\bot$	m_1	m_2	$\top$
m_1	m_1	m_1	$\top$	$\top$
m_2	m_2	$\top$	m_2	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

where $m_1 \neq m_2$

Definition 1 A *method* is a partial function mapping a single method name to a particular method body. We use the notation:

$$a \mapsto m$$

for the method that maps the name $a \in \mathcal{N}$ to the method body $m \in \mathcal{B}$.

Definition 2 A *method dictionary*, $d \in \mathcal{D}$ is a total function, $d : \mathcal{N} \to \mathcal{B}^\star$ that maps only a finite subset of method names to bodies, *i.e.*, where $d^{-1}(\mathcal{B})$ is finite, and $d^{-1}(\top) = \emptyset$.

Note that a method dictionary represents a finite set of methods. For this reason we will always specify them extensionally, listing only the mappings to elements in $\mathcal{B}$. For example,

$$d = \{a \mapsto m_1, b \mapsto m_2\}$$

defines a method dictionary d that maps method name a to body m_1 and b to m_2, and all other method names to $\bot$.

Definition 3 A *class*, $c \in \mathcal{C}$, is either the empty class, nil, or a sequence $\langle \alpha, d \rangle \cdot c'$, with attributes $\alpha \subset \mathcal{A}$, method dictionary $d \in \mathcal{D}$, and superclass $c' \in \mathcal{C}$.

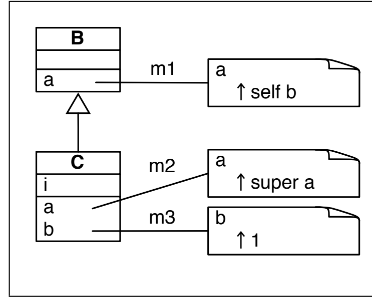


Fig. 9. An example hierarchy

For the purpose of explaining traits, we do not need to detail the behavior of methods. All that is necessary is to track self and super calls. We therefore model:

—*selfSends* : $\mathcal{B} \rightarrow 2^{\mathcal{N}}$, the set of method names used in self-sends, and
—*superSends* : $\mathcal{B} \rightarrow 2^{\mathcal{N}}$, the set of method names used in super-sends.

Note that it is considered poor style for a method to perform a super-send to a different method [Riel 1996]. We would therefore expect that for a given method $l \mapsto m$, *superSends*$(m) = \emptyset$ or $\{l\}$. However, since programming languages do not usually enforce this practice, we allow for the more general case.

We extend *selfSends* and *superSends* to sets of methods in the obvious way:

—*selfSends* : $2^{\mathcal{B}} \rightarrow 2^{\mathcal{N}}$, *selfSends*$(\mu) \stackrel{\text{def}}{=} \bigcup_{m \in \mu}$ *selfSends*(m)
—*superSends* : $2^{\mathcal{B}} \rightarrow 2^{\mathcal{N}}$, *superSends*$(\mu) \stackrel{\text{def}}{=} \bigcup_{m \in \mu}$ *superSends*(m)

(We will later further extend these functions to traits and classes in a similar way, simply taking the union of all self- or super-sends of the methods belonging to the trait or class.)

Example. Consider a class c defined as follows:

$$c = \langle \{i\}, \{a \mapsto m_2, b \mapsto m_3\} \rangle \cdot \langle \emptyset, \{a \mapsto m_1\} \rangle \cdot \mathsf{nil}$$

Class c has attribute i and methods $a \mapsto m_2$ and $b \mapsto m_3$. Its superclass is $\langle \emptyset, \{a \mapsto m_1\} \rangle \cdot \mathsf{nil}$, whose superclass in turn is nil. (For conciseness, we will in future omit the trailing ·nil from all examples.) Note that $a \mapsto m_2$ in c overrides $a \mapsto m_1$ in its superclass (see Figure 9).

Suppose now that:

$$\begin{array}{rclrcl} \textit{selfSends}(m_1) & = & \{b\} & \textit{superSends}(m_1) & = & \emptyset \\ \textit{selfSends}(m_2) & = & \emptyset & \textit{superSends}(m_2) & = & \{a\} \\ \textit{selfSends}(m_3) & = & \emptyset & \textit{superSends}(m_3) & = & \emptyset \end{array}$$

Since the superclass of c requires, but does not implement b, we also see that method b is abstract. Furthermore, $a \mapsto m_2$ not only overrides $a \mapsto m_1$, but also makes use of m_1 by means of a super call. Thus we see that knowledge of the superclass chain is essential to capturing the semantics of classes, since the meaning of $a \mapsto m_2$ depends on $a \mapsto m_1$.

Shortly we will formalize what it means for a class to be abstract or concrete, and when two classes constructed in different ways are equivalent.

4.2 Traits

We model traits as an extension of method dictionaries where some methods may conflict. Conflicts may arise when traits are composed; the conflicts can be resolved when the composed trait is used in another class or trait. A trait both provides methods (*i.e.*, the methods implemented in the trait) and requires methods (*i.e.*, those that are invoked by self-sends and super-sends, but are not provided).

Definition 4 A *trait*, $t \in \mathcal{T}$, is a function, $t : \mathcal{N} \rightarrow \mathcal{B}^\star$, where $t^{-1}(\mathcal{B} \cup \{\top\})$ is finite.

Example. A trait, like a method dictionary, represents a finite set of methods. For example,

$$t_1 = \{a \mapsto m_1, b \mapsto m_2, c \mapsto \top\}$$

defines a trait t that maps method name a to body m_1, b to m_2, and for which method name c has a conflict. (Assume m_1 and m_2 to be the same method bodies we saw in the previous example.)

Since traits are just finite mappings, two traits are equal when these mappings are equal, that is, when equal names map to equal method bodies. (Equality of method bodies may be established in a variety of ways, *e.g.*, by the location of the source code, or by the syntactic equality or equivalence of source code.)

By convention, *selfSends* and *superSends* of $\top$ and $\bot$ are all $\emptyset$. We extend *selfSends* and *superSends* to traits in the obvious way:

—*selfSends* : $\mathcal{T} \rightarrow 2^{\mathcal{N}}$, *selfSends*$(t) \stackrel{\text{def}}{=} \bigcup_{l \in \mathcal{N}}$ *selfSends*(l)

—*superSends* : $\mathcal{T} \rightarrow 2^{\mathcal{N}}$, *superSends*$(t) \stackrel{\text{def}}{=} \bigcup_{l \in \mathcal{N}}$ *superSends*(l)

In the example, *selfSends*$(t_1) = \{b\}$ and *superSends*$(t_1) = \{a\}$.

Definition 5 The *conflicts* : $\mathcal{T} \rightarrow 2^{\mathcal{N}}$, of a trait t are defined by:

$$\textit{conflicts}(t) \stackrel{\text{def}}{=} \{l \mid t(l) = \top\}$$

In the example, *conflicts*$(t_1) = \{c\}$.

Note that every method dictionary is, by definition, a trait, but traits with conflicts are not method dictionaries. In fact, a method dictionary $d \in \mathcal{D}$ is just a *conflict-free* trait, that is, a trait d such that *conflicts*$(d) = \emptyset$. We therefore consider that $\mathcal{D} \subset \mathcal{T}$.

The names of the methods defined for a trait are those that it provides. A trait may also require a set of methods that parameterize the provided behavior.

Definition 6 The *provided* method names, *provided* : $\mathcal{T} \rightarrow 2^{\mathcal{N}}$, of a trait t are:

$$\textit{provided}(t) \stackrel{\text{def}}{=} t^{-1}(\mathcal{B})$$

i.e., the set of all names that t does not map to $\bot$ or $\top$. In the example, *provided*$(t_1) = \{a, b\}$.

Definition 7 The *required* names, *required* : $\mathcal{T} \rightarrow 2^{\mathcal{N}}$, of a trait t are:

$$\textit{required}(t) \stackrel{\text{def}}{=} \textit{selfSends}(t) \setminus \textit{provided}(t)$$

In the example (2), *required*$(t_1) = \emptyset$, since $\{b\} \setminus \{a, b\} = \emptyset$. In contrast, if we have a trait $t' = \{a \mapsto m_1\}$, then *required*$(t') = \{b\}$ since b is sent to self but not provided.

Notice that the required names of a trait do not consider super-sends. This is because traits, like mixins, do not bind super. When we compose classes from traits, we will take super-sends into account when we ascertain whether the class is well-founded (cf. Definition 19).

Since traits contain only methods, they cannot specify any state, nor can they access state directly. Trait methods can access state *indirectly*, using required methods that are ultimately provided by accessors (getter and setter methods) in a class that uses the trait.

4.3 Composing Classes from Traits

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 summarize this relationship with the "equation"

$$\textit{Class} = \textit{Superclass} + \textit{State} + \textit{Traits} + \textit{Glue methods}$$

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

Whereas an ordinary class has the form $\langle \alpha, d \rangle \cdot c'$, a class composed from traits has the form

$$\langle \alpha, d \rhd t \rangle \cdot c'$$

where t is a trait, and d is a method dictionary that may extend and override t. In general, t can be a *composition clause*, an expression that specifies the sum of several traits, and possibly aliases or excludes selected methods. The glue we refer to consists precisely of the overriding, aliasing and exclusion operations.

Definition 8 The *sum* of two traits is formed by taking the union of the non-conflicting methods and disabling the conflicting methods. For traits t_1 and t_2, we define their sum $(t_1 + t_2) : \mathcal{N} \to \mathcal{B}^\star$ as follows:

$$(t_1 + t_2)(l) \stackrel{\text{def}}{=} t_1(l) \sqcup t_2(l)$$

For example:

$$\{a \mapsto m_1, b \mapsto m_2, c \mapsto m_3\} + \{a \mapsto m_1, b \mapsto m_4\} = \{a \mapsto m_1, b \mapsto \top, c \mapsto m_3\}$$

Proposition 1 Trait sum is associative and commutative.

PROOF. Immediate from the definition, since the *join* operator $\sqcup$ is associative and commutative. □

Because trait sum is commutative, conflicts must be resolved explicitly (cf. Section 4.5). Note that equal methods do not conflict, so in the previous example there is no conflict for a.

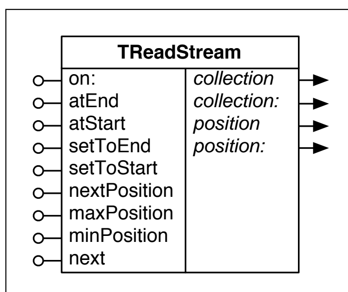


Fig. 10. The trait TReadStream with provided and required methods

Definition 9 A method dictionary d may *override* some of the methods in a trait t. We define $d \triangleright t : \mathcal{N} \rightarrow \mathcal{B}^{\star}$ as follows:

$$(d \triangleright t)(l) \stackrel{\text{def}}{=} \begin{cases} t(l) & \text{if } d(l) = \bot \\ d(l) & \text{otherwise} \end{cases}$$

Overriding is the key mechanism for resolving conflicts. Note that $d \triangleright t$ is, in general, a trait, not a method dictionary. However, even if t contains conflicts, we can always choose d so that $d \triangleright t$ will be conflict-free. For example,

$$\{b \mapsto m_2\} \triangleright \{a \mapsto m_1, b \mapsto \top, c \mapsto m_3\} = \{a \mapsto m_1, b \mapsto m_2, c \mapsto m_3\}$$

In class definitions of the form $\langle \alpha, d \triangleright t\rangle \cdot c'$, d will typically be used to resolve conflicts in t, and to provide any missing methods required by t. For the moment, we will assume that a class formed in this way is *well-defined*, and defer a discussion of what this means to Section 4.6.

The flattening property. An important property follows from the way that classes are constructed from traits. If $c = \langle \alpha, d \triangleright t\rangle \cdot c'$ is well-defined (cf. Section 4.6), and, in particular, if $d' = d \triangleright t$ is conflict-free, then c can clearly be *flattened* to an equivalent definition $c = \langle \alpha, d'\rangle \cdot c'$ that does not make use of traits. In other words, in any class defined using traits, the *traits can be inlined* to give an equivalent class definition that does not use traits.

As a consequence, traits and classes have the following properties.

—*Methods defined in the class take precedence over methods provided by a trait.* This follows from the fact that the methods in the method dictionary d of the class override those provided by t in $d \triangleright t$.

—*Trait methods take precedence over superclass methods.* This follows from the flattening property. Since, $\langle \alpha, d \triangleright t\rangle \cdot c'$ can be *flattened* to $\langle \alpha, d'\rangle \cdot c'$, trait methods behave as if they were implemented in the class itself.

—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.

Note that the flattening property expresses only that the use of traits can be flattened within a class definition. It does *not* state that inheritance between classes can be flattened. In general inheritance hierarchies can be *refactored*, but not flattened. Later, in Section 4.7, we will consider when classes that have been refactored using traits are equivalent.

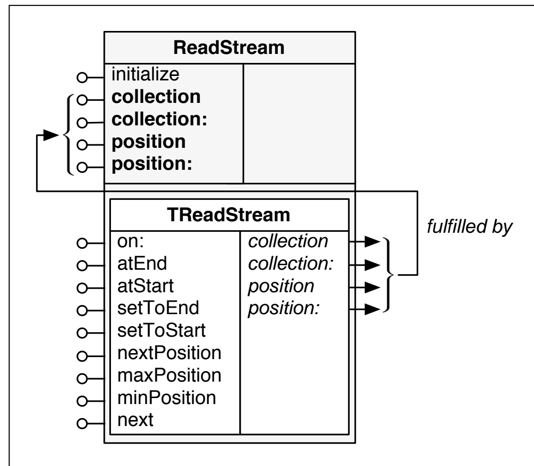


Fig. 11. The class ReadStream composed from the trait TReadStream

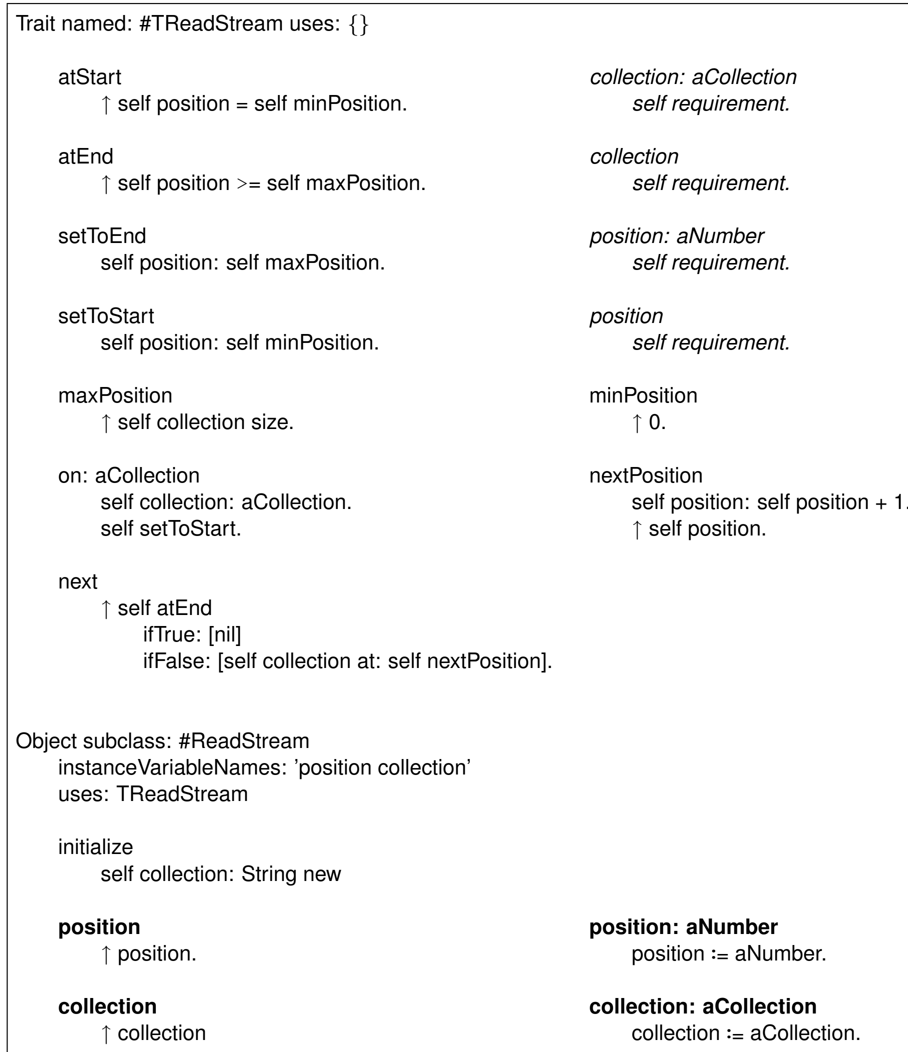
Running Example. Suppose that we want to implement a library that provides streams which may be readable, writeable, both readable and writeable, or synchronized. For clarity, trait names start with the letter T, and class names do not. We *italicize* required methods and **embolden** glue methods. Because traits have been implemented in the Smalltalk dialect Squeak [Ingalls et al. 1997], we present the code in Smalltalk. Note that in the following examples (as in the other examples shown in this paper), we do not address the question of whether the composition of traits is well-behaved, but simply whether it is well-formed (*i.e.*, whether it will compile).

The basic idea is to build the stream classes in a class library from elementary traits such as TReadStream, TWriteStream and TSynchronize. We introduce a minor extension to UML to present traits graphically, as seen in Figure 10. The left column lists the provided methods of TReadStream and the right column lists the required methods. The code implementing this trait is shown below. Required methods (shown in *italics*) are flagged by the use of the method body self requirement.

As illustrated in Figure 11 and the corresponding Smalltalk code in Figure 12, we create the class ReadStream by using the trait TReadStream, which is parameterized by the required methods collection, collection:, position, and position:. To be complete, the class ReadStream has to fulfill these requirements by providing corresponding glue methods. In the example, the methods collection, position, and position: are implemented as accessors to two instance variables collection and position, while the method collection: both enables a stream to adopt a new collection, and ensures that the stream is correctly positioned at its start. ReadStream also implements a method for initializing the instance variables.

4.4 Composite 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. Here too, the composition order is not important, and methods defined in a composite trait take precedence over the methods



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

    atStart
        ↑ self position = self minPosition.

    atEnd
        ↑ self position >= self maxPosition.

    setToEnd
        self position: self maxPosition.

    setToStart
        self position: self minPosition.

    maxPosition
        ↑ self collection size.

    on: aCollection
        self collection: aCollection.
        self setToStart.

    next
        ↑ self atEnd
            ifTrue: [nil]
            ifFalse: [self collection at: self nextPosition].

    collection: aCollection
        self requirement.

    collection
        self requirement.

    position: aNumber
        self requirement.

    position
        self requirement.

    minPosition
        ↑ 0.

    nextPosition
        self position: self position + 1.
        ↑ self position.


Object subclass: #ReadStream
    instanceVariableNames: 'position collection'
    uses: TReadStream

    initialize
        self collection: String new

    position
        ↑ position.

    position: aNumber
        position := aNumber.

    collection
        ↑ collection

    collection: aCollection
        collection := aCollection.
```

Fig. 12. Smalltalk implementation of the class ReadStream and the trait TReadStream

of its subtraits.

Definition 10 A *composite trait* is a trait expression of the form $d \triangleright t$, where $d \in \mathcal{D}$ and t is a composition clause (trait expression) using only trait sum ($+$), aliasing ($\rightarrow$) and exclusion ($-$) operators (cf. Section 4.5).

Even in the case of multiple levels of composition, the flattening property remains valid. The semantics of a method do not depend on whether it is defined in a trait or in entities that use the trait (cf. Section 5.2).

Example. Since the traits TReadStream and TWriteStream contain several identical methods, we factor out the duplicated behavior into a new trait TPositionableStream, which

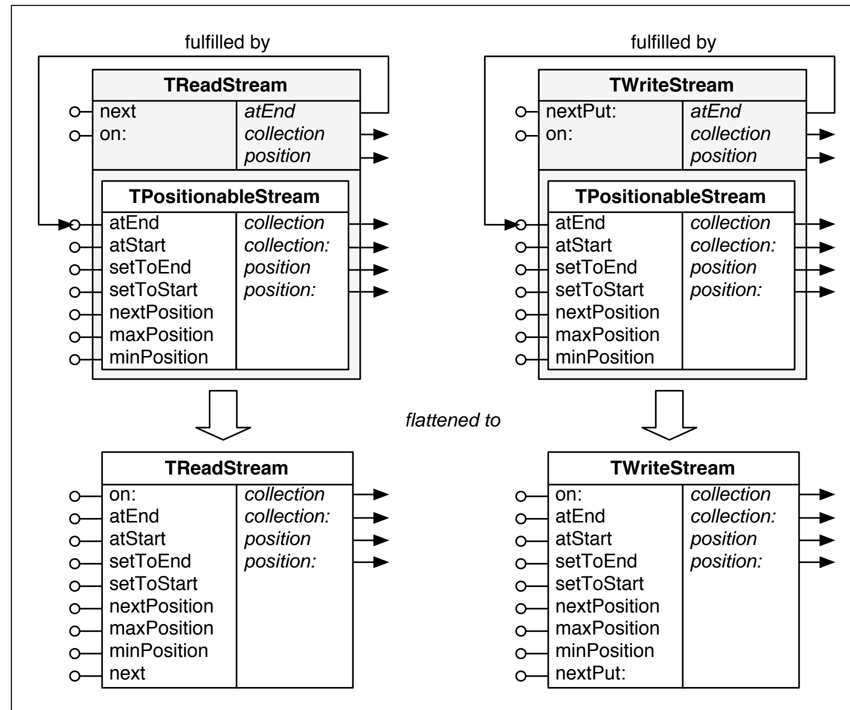


Fig. 13. TReadStream and TWriteStream as composite traits

provides the functionality to manipulate a position over a collection. As illustrated in Figure 13, the traits TReadStream and TWriteStream can then be expressed in terms of the trait TPositionableStream.

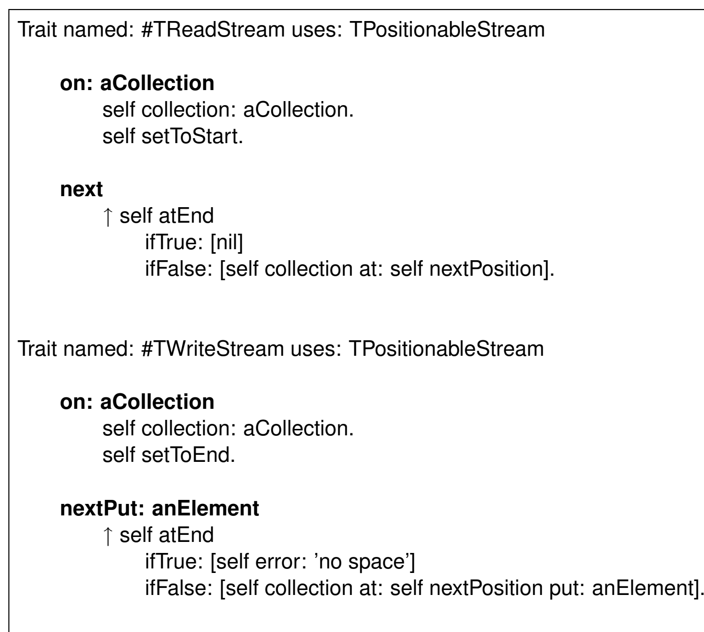
The implementation of TPositionableStream is identical to the implementation of TReadStream (cf. Figure 12) without the methods next and on:. Figure 14 shows the implementation of the traits TReadStream and TWriteStream, which both use the trait TPositionableStream. The trait TReadStream overrides the methods on: and next, which position the stream to the beginning of the given collection and read the next element. Similarly, the trait TWriteStream overrides the methods on: and nextPut:, which position the stream to the end of the given collection and append an element.

Note that the unfulfilled requirements of TPositionableStream are propagated to the traits TReadStream and TWriteStream, respectively. This means that the traits TReadStream and TWriteStream are also parameterized by the required methods collection, collection:, position, and position:.

4.5 Conflict Resolution

A conflict arises if and only if we compose two traits that provide identically named methods with different bodies.[5] In particular, this means that if the *same* method is obtained more than once via different paths, there is no conflict (cf. Definition 8 and Section 5.2).

[5]In the Squeak implementation bodies are considered to be different if they originate from different traits—other strategies could also be adopted, such as comparing the source code, or the bytecode.



```
Trait named: #TReadStream uses: TPositionableStream

    on: aCollection
        self collection: aCollection.
        self setToStart.

    next
        ↑ self atEnd
            ifTrue: [nil]
            ifFalse: [self collection at: self nextPosition].


Trait named: #TWriteStream uses: TPositionableStream

    on: aCollection
        self collection: aCollection.
        self setToEnd.

    nextPut: anElement
        ↑ self atEnd
            ifTrue: [self error: 'no space']
            ifFalse: [self collection at: self nextPosition put: anElement].
```

Fig. 14. Implementation of the composite traits TReadStream and TWriteStream

Method conflicts must be resolved explicitly by defining an overriding method in the class or in the composite trait, or by means of exclusion in the composition clause. This guarantees that the conflict can be resolved only on the level of the composite, but not by another subtrait that happens to provide a method with the same name.

To provide a way for an overriding method to access a conflicting method, and thereby avoid code duplication, traits support *aliasing*. Aliases allow the programmer to make a trait method available under another name, and are very useful if the original name is excluded by a conflict.

Definition 11 *Aliasing* introduces an additional name for an existing method:

$$t[a \rightarrow b](l) \stackrel{\text{def}}{=} \begin{cases} t(l) & \text{if } l \neq a \\ t(b) & \text{if } l = a \text{ and } t(a) = \bot \\ \top & \text{otherwise} \end{cases}$$

For example:

$$\{a \mapsto m_1, b \mapsto m_2\}[c \rightarrow b] = \{a \mapsto m_1, b \mapsto m_2, c \mapsto m_2\}$$

Note that $\{a \mapsto m_1, b \mapsto m_2\}[a \rightarrow b] = \{a \mapsto \top, b \mapsto m_2\}$, which expresses that an attempt to alias a method name that is already bound will introduce a conflict.

Aliases are discussed further in Section 5.2.

In addition to overriding and aliasing, trait composition also support *exclusion*, which allows a programmer to exclude methods that would otherwise be provided by a trait, and thus to avoid a conflict before it occurs.

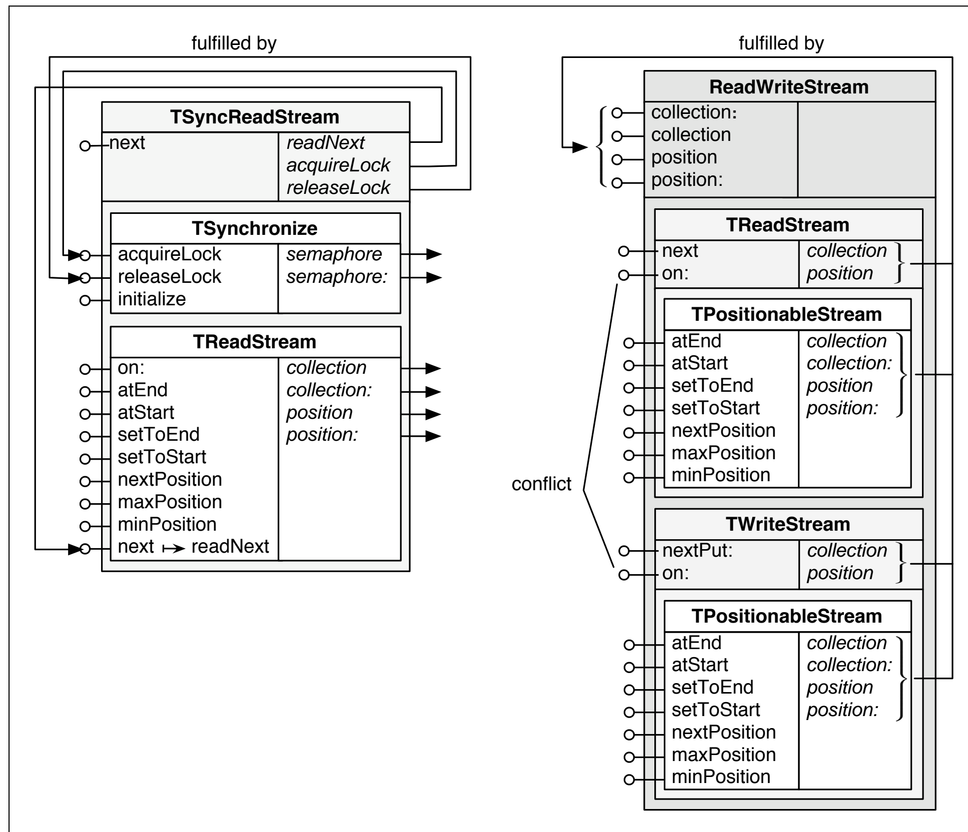


Fig. 15. Compositions with aliases and conflicts

Definition 12 *Exclusion* removes a method from a trait:

$$(t - a)(l) \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } a = l \\ t(l) & \text{otherwise} \end{cases}$$

For example:

$$\{a \mapsto m_1, b \mapsto \top\} - b = \{a \mapsto m_1\}$$

Example. As a concrete example for the use of aliases, consider the trait TSyncReadStream in Figure 15 (left). This trait represents a synchronized read stream, and it is constructed as the composition of the traits TSynchronize and TReadStream. To ensure that element access is properly synchronized, the TSyncReadStream redefines the next method provided by the trait TReadStream. Since this redefinition needs to invoke the original next method provided by the trait TReadStream, we create an alias that makes the original method TReadStream>>next available under the new name readNext.

Figure 16 shows the actual implementation of the traits TSyncReadStream and TSynchronize. (The implementation of TReadStream is shown in Figure 14.) The implementation of trait TSynchronize is straightforward; it provides the methods acquireLock, releaseLock, and initialize, while it requires the methods semaphore: and semaphore.

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

    acquireLock                                  semaphore
        self semaphore wait.                         self requirement.

    initialize                                   semaphore: aSemaphore
        self semaphore: Semaphore new.               self requirement.
        self releaseLock.

    releaseLock
        self semaphore signal.


Trait named: #TSyncReadStream uses: TSynchronize + (TReadStream @ {#readNext -> #next})

    next
        | read |
        self acquireLock.
        read := self readNext.
        self releaseLock.
        ↑ read.
```

Fig. 16. Implementation of the traits TSynchronize and TSyncReadStream

The implementation of trait TSyncReadStream is more interesting. In the composition clause of TSyncReadStream, we first use the operator @ to create the alias readNext for the method next provided by TReadStream, and then we use the operator + to compose the aliased trait with the trait TSynchronize. The method next is then overridden in TSyncReadStream so that it acquires a lock, calls the original method via the alias, and then releases the lock. Note that TSyncReadStream does not satisfy the requirements of the traits TReadStream and TSynchronize, which means that they are propagated and become requirements of TSyncReadStream itself.

As an example of a conflict, consider the class ReadWriteStream shown in Figure 15 (right). This class is built from the two traits TReadStream and TWriteStream, which each provide their own version of the method on:. This results in a conflict that may be resolved by excluding one of the conflicting methods or by overriding it in the composite class.

In our example, we avoid the conflict by excluding the method TReadStream>>on:, which means that the method TWriteStream>>on: will be included in the composite. The corresponding composition clause uses the exclusion operator:

```
Stream subclass: #ReadWriteStream
    uses: (TReadStream − {#on:}) + TWriteStream
```

As shown in Figure 13, the traits TReadStream and TWriteStream are both composed from the trait TPositionableStream. Thus, all methods originating from TPositionableStream are *identical* in both traits and do not create conflict (cf. Definition 8).

4.6 Well-definedness

We have deferred a discussion of when a class built from traits is well-defined. We will now make that notion precise. In particular, we will define what it means for a class to be

valid and well-founded.

Definition 13 The *dictionary* of a class c, $\mathit{dict}(c)$, is the $\triangleright$ composition of the (flattened) method dictionaries in its inheritance chain:

$$\mathit{dict}(c) \stackrel{\text{def}}{=} \begin{cases} \{\} & \text{if } c = \mathsf{nil} \\ d \triangleright \mathit{dict}(c') & \text{if } c = \langle \alpha, d \rangle \cdot c' \end{cases}$$

Note that the *dictionary* of a class c is not necessarily a (conflict-free) *method dictionary*: if c is composed from traits, *i.e.*, $c = \langle \alpha, d \triangleright t \rangle \cdot c'$, then $d \triangleright t$ is not necessarily a (conflict-free) method dictionary in $\mathcal{D}$, and $\mathit{dict}(c)$ might contain conflicts. We therefore need the following definition to tell us when a class composed from traits is *valid*:

Definition 14 A class c is *valid* if $\mathit{conflicts}(\mathit{dict}(c)) = \emptyset$.

Definition 15 The *method lookup*, $c \gg a$, of a method name a in a class c is:

$$c \gg a \stackrel{\text{def}}{=} \mathit{dict}(c)(a)$$

Definition 16 The *provided* names, $\mathit{provided} : \mathcal{C} \to 2^{\mathcal{N}}$, of a class c are:

$$\mathit{provided}(c) \stackrel{\text{def}}{=} \{ l \in \mathcal{N} \mid c \gg l \in \mathcal{B} \}$$

i.e., the set of all names that $\mathit{dict}(c)$ does not map to $\bot$ or $\top$.

Definition 17 The *increment*, $\mathit{delta}(c)$, of a class c, is:

$$\mathit{delta}(c) \stackrel{\text{def}}{=} \begin{cases} \{\} & \text{if } c = \mathsf{nil} \\ d & \text{if } c = \langle \alpha, d \rangle \cdot c' \end{cases}$$

Definition 18 The *superclass*, $\mathit{super}(c)$, of c is:

$$\mathit{super}(c) \stackrel{\text{def}}{=} \begin{cases} \mathsf{nil} & \text{if } c = \mathsf{nil} \\ c' & \text{if } c = \langle \alpha, d \rangle \cdot c' \end{cases}$$

Definition 19 A class c is *well-founded* if and only if all super-sends in its inheritance chain are bound, *i.e.*, if $\mathit{superSends}(\mathit{delta}(c)) \subseteq \mathit{provided}(\mathit{super}(c))$ and $\mathit{super}(c)$ is well-founded. nil is well-founded by convention.

For a particular programming language, a class that is not well-founded may generate run-time errors or compile-time errors, depending on the philosophy of its designers.

Definition 20 A class c is *well-defined* if c is valid and well-founded.

Now we can state the *flattening property* more precisely:

Proposition 2 If $c = \langle \alpha, d \triangleright t \rangle \cdot c'$ is well-defined and $d' = d \triangleright t$, then $c = \langle \alpha, d' \rangle \cdot c'$ is an equivalent, flattened definition of c.

PROOF. Follows trivially from definitions 13 and 14. Since c is well-defined, it is valid, and hence $d' \triangleright \mathit{dict}(c')$ is conflict-free, so $\langle \alpha, d' \rangle \cdot c'$ is an equivalent, valid class definition. $\square$

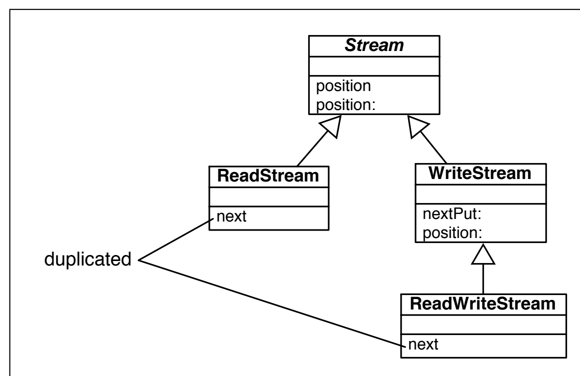


Fig. 17. Code duplication in the Smalltalk Stream hierarchy

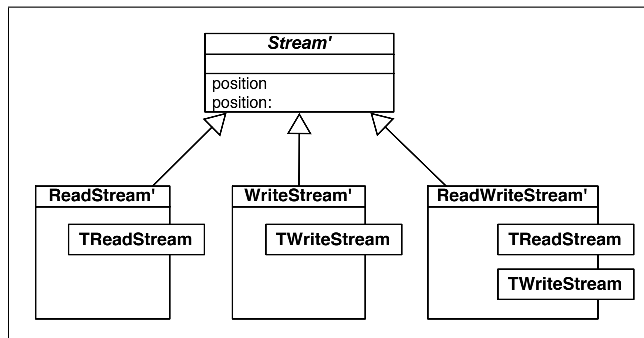


Fig. 18. The Smalltalk Stream hierarchy refactored using traits

4.7 Refactoring, Reachability and Equivalence

The flattening property helps us to ensure that the semantics of a class do not change when it is rewritten as a composition of traits. But it is inadequate for reasoning about the equivalence of classes when an entire class hierarchy is refactored. The reason for this is that flattening says nothing about which methods are reachable through the inheritance chain of the class by means of self- and super-sends.

Consider, for example, the class hierarchy in Figure 17. We can represent this formally as follows (eliding the trailing nil):

$$
\begin{aligned}
\mathsf{Stream} &= \langle \alpha, \{\mathsf{position} \mapsto m_{position}, \mathsf{position\!:} \mapsto m_{position:}\}\rangle \\
\mathsf{ReadStream} &= \langle \beta, \{\mathsf{next} \mapsto m_{next}\}\rangle \cdot \mathsf{Stream} \\
\mathsf{WriteStream} &= \langle \gamma, \{\mathsf{nextPut\!:} \mapsto m_{nextPut:}, \mathsf{position\!:} \mapsto m_{position:'}\}\rangle \cdot \mathsf{Stream} \\
\mathsf{ReadWriteStream} &= \langle \beta, \{\mathsf{next} \mapsto m_{next}\}\rangle \cdot \mathsf{WriteStream} \\
\mathit{selfSends}(m_{next}) &= \{\mathsf{position}\} \\
\mathit{selfSends}(m_{nextPut:}) &= \{\mathsf{position}, \mathsf{position\!:}\} \\
\mathit{selfSends}(m) &= \{\} && \text{for all other values of } m \\
\mathit{superSends}(m_{position:'}) &= \{\mathsf{position\!:}\} \\
\mathit{superSends}(m) &= \{\} && \text{for all other values of } m
\end{aligned}
$$

One problem with this hierarchy is that the method next $\mapsto m_{next}$ is duplicated: it is implemented in both the class ReadStream and ReadWriteStream. We avoid this duplication by refactoring this hierarchy with traits as shown in Figure 18. In the refactored hierarchy, the classes ReadStream′, WriteStream′, and ReadWriteStream′ are direct subclasses of the class Stream′, while the actual read and write behavior is factored out into two traits TReadStream and TWriteStream.

Formally, the new hierarchy is expressed as:

$$\begin{aligned}
\mathsf{TReadStream} &= \{\mathsf{next} \mapsto m_{next}\} \\
\mathsf{TWriteStream} &= \{\mathsf{nextPut:} \mapsto m_{nextPut:}, \mathsf{position:} \mapsto m_{position:'}\} \\
\mathsf{ReadStream}' &= \langle \beta, \mathsf{TReadStream} \rangle \cdot \mathsf{Stream} \\
\mathsf{WriteStream}' &= \langle \gamma, \mathsf{TWriteStream} \rangle \cdot \mathsf{Stream} \\
\mathsf{ReadWriteStream}' &= \langle \beta \cup \gamma, \mathsf{TReadStream} + \mathsf{TWriteStream} \rangle \cdot \mathsf{Stream}
\end{aligned}$$

But how do we know that this new hierarchy really preserves the semantics of these classes? In general, we must consider not only the mapping from provided method names to method bodies, but also methods that may be reached by super-sends from those method bodies, methods that may be reached by super-sends from those methods, and so on. We therefore introduce the notation $c{\uparrow}a_1a_2...a_n = \langle m, c' \rangle$ to mean that it is possible for the method body bound to a_1 in class c to perform a super-send to a_2, and for the method body bound to a_2 in this context to perform a super-send, and so on, until eventually a_n can be super-sent. If this occurs, then a_n will be bound to method body m obtained from c'.

Definition 21 $c{\uparrow}\bar{a}$, where $c \in \mathcal{C}$ and $\bar{a} \in \mathcal{N}^+$, is defined recursively, as follows:

$$\begin{aligned}
\mathsf{nil}{\uparrow}a &\stackrel{\text{def}}{=} \langle \bot, \mathsf{nil} \rangle \\
c{\uparrow}a &\stackrel{\text{def}}{=} \begin{cases} \langle m, c \rangle & \text{if } m = \mathit{delta}(c)(a) \in \mathcal{B} \\ \mathit{super}(c){\uparrow}a & \text{otherwise} \end{cases} \\
c{\uparrow}\bar{a}b &\stackrel{\text{def}}{=} \begin{cases} \mathit{super}(c'){\uparrow}b & \text{if } c{\uparrow}\bar{a} = \langle m, c' \rangle \text{ and } b \in \mathit{superSends}(m) \\ \langle \bot, \mathsf{nil} \rangle & \text{otherwise} \end{cases}
\end{aligned}$$

For example,

$$\begin{aligned}
\mathsf{ReadWriteStream}{\uparrow}\mathsf{position:} &= \langle m_{position:'}, \mathsf{WriteStream} \rangle \\
\mathsf{ReadWriteStream}{\uparrow}\mathsf{position:\ position:} &= \langle m_{position:}, \mathsf{Stream} \rangle
\end{aligned}$$

For convenience, we also introduce the notation $c{\restriction}\bar{a}$, which returns just the method body reachable by $\bar{a}$ without its class.

Definition 22 A method body $m \in \mathcal{B}$ is *reachable* from class c, if $\exists \bar{a} \in \mathcal{N}^+$ such that $m = c{\restriction}\bar{a}$, where

$$c{\restriction}\bar{a} \stackrel{\text{def}}{=} m, \textit{ where } \exists c' \textit{ such that } c{\uparrow}\bar{a} = \langle m, c' \rangle$$

For example:

$$\begin{aligned}
\mathsf{ReadWriteStream}{\restriction}\mathsf{position:} &= m_{position:'} \\
\mathsf{ReadWriteStream}{\restriction}\mathsf{position:\ position:} &= m_{position:}
\end{aligned}$$

As expected, method bodies reachable without any super-sends correspond precisely to method lookups:

Proposition 3 For any class c and any single message a, $c\upharpoonright a = c \gg a$

PROOF. By induction on the depth of the inheritance hierarchy.
For the base case, $c = \mathsf{nil}$, we trivially have $\forall a, \mathsf{nil}\upharpoonright a = \bot = nil \gg a$.
Now consider $c = \langle \alpha, d\rangle \cdot c'$
a) Suppose $d(a) = m \in \mathcal{B}$. Then $c\upharpoonright a = m = c \gg a$.
b) Suppose $d(a) = \bot$. Then, by definition, $c\upharpoonright a = c'\upharpoonright a$. Similarly $c \gg a = c' \gg a$. But by induction, we have $c'\upharpoonright a = c' \gg a$, hence $c\upharpoonright a = c \gg a$ □

Definition 23 The reachability set of a class c, is:

$$\textit{reachable}(c) \stackrel{\text{def}}{=} \{\langle \bar{a}, c\upharpoonright \bar{a}\rangle \mid \bar{a} \in \mathcal{N}^+, c\upharpoonright \bar{a} \neq \bot\}$$

This precisely expresses which method bodies are reachable by means of self- and super-sends through the public methods of a class. For example:

$$\begin{aligned}
\textit{reachable}(\mathsf{ReadStream}) &= \{ \langle \mathsf{position}, m_{position}\rangle, \langle \mathsf{position:}, m_{position:}\rangle, \\
&\qquad \langle \mathsf{next}, m_{next}\rangle \} \\
\textit{reachable}(\mathsf{ReadWriteStream}) &= \{ \langle \mathsf{position}, m_{position}\rangle, \langle \mathsf{position:}, m_{position:'}\rangle, \\
&\qquad \langle \mathsf{position:position:}, m_{position:}\rangle, \\
&\qquad \langle \mathsf{next}, m_{next}\rangle, \langle \mathsf{nextPut:}, m_{nextPut:}\rangle \}
\end{aligned}$$

Two classes are equivalent if exactly the same method bodies are reachable by the same super-send chains.

Definition 24 A class c is *equivalent* to a class c', $c \equiv c'$, iff:

$$\textit{reachable}(c) = \textit{reachable}(c')$$

(Note that $\equiv$ is trivially reflexive, symmetric and transitive, so is, in fact, an equivalence.)

Proposition 4 $c \equiv c' \Rightarrow \textit{provided}(c) = \textit{provided}(c')$

PROOF. $a \in \textit{provided}(c) \Rightarrow c \gg a \in \mathcal{B} \Rightarrow c\upharpoonright a \in \mathcal{B} \Rightarrow \langle a, c\upharpoonright a\rangle \in \textit{reachable}(c) \Rightarrow \langle a, c\upharpoonright a\rangle \in \textit{reachable}(c') \Rightarrow \ldots \Rightarrow a \in \textit{provided}(c')$ □

In the example, it is now straightforward to show that $\mathsf{ReadStream} \equiv \mathsf{ReadStream}'$, and so on.

Finally, we would like to know which classes are abstract and which are concrete. To determine this, we must establish the set of all self-sends in the reachable method bodies, and check if these methods are actually provided. Those that are missing are required.

Definition 25 The set of self-sends, *selfSends*(c), of a class c is:

$$\textit{selfSends}(c) \stackrel{\text{def}}{=} \bigcup\{\textit{selfSends}(m) \mid \exists \bar{a}, \langle \bar{a}, m\rangle \in \textit{reachable}(c)\}$$

For example, $\textit{selfSends}(\mathsf{ReadWriteStream}) = \{\mathsf{position}, \mathsf{position:}\}$.

Definition 26 The set of *required* names, *required*(c), of a class c is:

$$\textit{required}(c) \stackrel{\text{def}}{=} \textit{selfSends}(c) \setminus \textit{provided}(c)$$

Definition 27 A class, $c \in \mathcal{C}$, is *concrete* if *required*$(c) = \emptyset$. A class that is not concrete is *abstract*.

In particular, ReadWriteStream is concrete since

$$\textit{selfSends}(\mathsf{ReadWriteStream}) \subset \textit{provided}(\mathsf{ReadWriteStream})$$

so *required*$(\mathsf{ReadWriteStream}) = \emptyset$.

Note that a class that is built-up from traits need not be concrete to be well-defined, so it is also possible to compose abstract classes from traits.

5. DISCUSSION AND EVALUATION

In this section, we evaluate traits with respect to the problems discussed in Section 2, and we discuss some decisions that significantly influenced the design of traits and the trait operations. Our main concerns are reusability and the understandability of programs written using traits.

5.1 Evaluation Against the Identified Problems

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

Duplicated Features. Duplicated code can easily be factored out into unique traits, which may then be used to compose arbitrary classes, independent of their position in the class hierarchy [Black et al. 2003].

Inappropriate Hierarchies. Trait composition enables the reuse of behavior in a way that is complementary to single inheritance: with trait composition being the primary mechanism for (fine-grained) code reuse, the inheritance hierarchy is freed to capture conformance and conceptual relationships between classes. This means that the programmer can avoid inappropriate inheritance hierarchies by moving reusable methods into traits and apply them only to the classes where they are appropriate and actually needed.

This is illustrated in Figures 17 and 18, which show a part of the Smalltalk stream hierarchy constructed using single inheritance and traits, respectively. The traditional hierarchy without traits (Figure 17) does not correctly model the conceptual relationship between the stream classes: the class ReadWriteStream is related to WriteStream but not to ReadStream. Furthermore, this hierarchy involves code duplication. Both of these problems are avoided in the hierarchy based on traits (Figure 18). This hierarchy maximizes code reuse and is conceptually consistent.

Duplicated Wrappers. Generic wrappers, such as the synchronization wrappers discussed in Section 2.1, can be expressed easily with traits. In fact, the solution shown at the right side of Figure 1 would work if SyncReadWrite were a trait, since super in a trait refers to the superclass of the class that will actually use that trait. If SyncA is defined to be a subclass of A and SyncB a subclass of B, and both subclasses use the trait SyncReadWrite, the super-send in the trait's read and write: methods will be bound to the respective superclasses A and B *when the classes SyncA and SyncB are composed*. Other kinds of generic wrappers can be defined in much the same way.

Conflicting Features. Traits avoid state conflicts entirely, because traits cannot define state. Method conflicts may be resolved either by explicitly excluding one of the conflicting methods from the composition, or by overriding the conflict in the composite entity. In general, fewer conflicts arise with traits than with multiple inheritance, because in our experience traits tend to remain lean, focussing on a small set of collaborating features.

Lack of Control and Dispersal of Glue Code. One of the most significant differences between traits and mixins is that trait sum is associative and commutative, so the ordering of the composition is irrelevant. As a consequence, the composite entity is always in full control of the composition: for each conflicting feature, the composite entity can *independently* choose which trait should take precedence or how the available implementations should be composed. This avoids the need for intermediate "glue components" that are spread over the inheritance hierarchy. Instead, the glue code is always located in the composing entity, reflecting the idea that the composing entity is in complete control of plugging together the components that implement its aspects. This property nicely separates the glue code from the code that implements the different aspects, and it makes a class easy to understand, even if it is composed from many different parts.

As an illustration, reconsider the example discussed in Section 3.1, where we want to create a new class MyRectangle based on the class Rectangle and two components adding color and a border. With traits, this is done by putting the color and border behavior into two traits TColor and TBorder and then defining the new class MyRectangle as a subclass of Rectangle that uses these traits. Because the features of the traits are unordered and fully accessible from within the composite class MyRectangle, all the glue code necessary to resolve conflicts and obtain the intended behavior can be defined within the class MyRectangle.

Note that although trait composition is unordered, it can be productively combined with inheritance to obtain a large variety of different partially ordered compositions. The basic idea is that if we want a class C to use two traits T1 and T2 in that order, we first introduce a superclass C′ that uses T1, and then we define C to inherit from C′ and use T2. This has the consequence that the methods in T2 override the methods in T1. This strategy proved itself in practice when we refactored the Smalltalk collection hierarchy (see Section 7).

Fragile Hierarchies. Any hierarchical approach to composing software is bound to be fragile with respect to certain kinds of change: if a feature that is used by many clients changes, the change will clearly affect all the clients. The important questions are: how severely will the change affect the features and the correctness of direct and indirect clients? Do we need to change implementations, or only glue code? Will there be a ripple-effect throughout the entire hierarchy due to apparently innocuous changes? Are there changes that implicitly change the behavior of direct or indirect clients in unexpected ways?

Extending a trait so that it provides additional methods may well affect clients by introducing a new conflict. However, the design of trait composition (chiefly the commutativity of composition and explicit conflict resolution) means such changes cannot lead to implicit and unexpected changes in the behavior of direct or indirect clients. Furthermore, a direct client can generally resolve a conflict without changing or introducing any other traits, so no ripple effect will occur. For example, if a new method is added, a direct client can always reestablish its original behavior by excluding the newly added method. Neither additional traits, nor additional methods, nor changes to existing methods will be needed.

In contrast, adding a new method to a mixin may require introducing new glue mixins as well as glue methods in the composite entity in order to reestablish the original behavior (see Section 2.2).

Traits also avoid the fragility problem we identified in multiple inheritance languages such as C++ and Eiffel, where methods become cluttered with navigational glue code when a programmer resolves an ambiguity by explicitly naming the class that provides a certain method. With traits, conflicting features are accessed by aliases, which are defined in the composition clause and can be called like regular methods. By avoiding tangled class references in the source code, this approach leads to hierarchies that are more robust and easier to understand.

5.2 Design Decisions

Traits were designed with other models of classes and inheritance 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 are a new concept because they are composed using a set of distinct composition operators rather than by single inheritance, and because they cannot define state. Like mixins, they are finer-grained units of reuse than classes and are not tied to a specific place in the inheritance hierarchy. We believe that these two properties improve code reuse and enable better conceptual modeling. Fine-grained reuse is important because the gulf that lies between entire classes and individual methods is too wide.

Traits allow classes to be built by composing reusable behaviors rather than by implementing a large and unstructured set of methods. But, unlike mixins composition, trait composition is unordered; this agrees with the unordered nature of the methods in a class.

Single Inheritance and the Flattening Property. Instead of replacing single inheritance, we decided to keep this familiar concept and 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 attributes) that are available in a class. If a class can inherit from only 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 access to inherited features is very convenient, but it also assigns semantics to the place of a method in the inheritance hierarchy. Therefore, it is generally not possible to understand a class hierarchy without knowing in which class a certain method is implemented.

Traits operate at a finer granularity than inheritance; they are typically used to modularize the behavior *within* a class. As such, traits are designed to capture behavior but not state. In addition, trait composition attributes no semantic significance to the place where a method is defined, with the result that traits enjoy the flattening property.

In combination with single inheritance, traits and the flattening property provide a smooth migration path for single inheritance languages. Given appropriate tool support (see Section 6.2), a system based on traits not only allows one to write and execute traditional single inheritance code, but even if there are hundreds of deeply composed traits, the user can still *view and edit* the classes in the same way as if the system were implemented without using traits at all.

Aliasing. Many multiple inheritance implementations provide access to overridden features by requiring the programmer to explicitly name the defining superclass in the source code. C++ provides the scope operator (::) [Stroustrup 1997], whereas Eiffel provides the keyword Precursor [Meyer 1997]. With traits, we chose method aliasing in preference to placing 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 cause aspects of the trait structure to be hard-coded in the methods that use the the traits. 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. It works with the flattening property because the flattening process can simply introduce a new name for the aliased method body.

Although there are some similarities between aliasing and method renaming as provided by Eiffel, there are also essential differences. Defining an alias y for a method x in the trait T just establishes an alternative name y without affecting the original one. In particular, all references to the original name x in the used trait T remain unchanged (*i.e.*, they still refer to the original name x). In contrast, when a method x is renamed to y in an Eiffel class C, the original method name x becomes undefined, and all the references to x in the class C are changed so that they conceptually refer to the new method name y.

While renaming violates the flattening property, it has the advantage that it completely frees the old name x as if it were never used in C (see the discussion of unintended name clashed below). Furthermore, renaming works well with recursive methods, where aliasing is not really adequate.

Unintended Name Clashes. With traits, as with any other name-based approach to composing software features, unintentional naming conflicts may arise. For example, consider a Java class that should implement two interfaces, but where each of these interfaces specifies a method with precisely the same name (and signature), and yet with different semantics. The same problem also appears in many mixin approaches such as Strongtalk [Bak et al. 2002] and JAM [Ancona et al. 2000]: if two mixins provide or require two semantically different methods that happen to have the same name, they cannot easily be composed.

At present, traits offer no real solution to this problem — when two traits are composed, it may be that each requires a semantically different method that happens to have the same name. Unlike Eiffel's renaming, aliases alleviate the problem only to a small extent. In our view, a complete solution requires good refactoring tools, or, preferably, a flexible encapsulation mechanism [Schärli et al. 2004].

Conflict Resolution Strategies. Although traits are based on single inheritance, a form of diamond problem may arise when features from the same trait are obtained multiple times via different paths. For example, consider the trait ReadWriteStream (Figure 15), which uses two traits TWriteStream and TReadStream, which in turn both use the trait TPositionableStream (Figure 13).

Since traits contain no state, the most nefarious diamond problem does not arise. Nevertheless, in our example, a method atEnd provided by TPositionableStream will be obtained by ReadWriteStream twice. The key language design question is: should this be considered a conflict?

As established in Definition 8, there is no conflict if the *same* method is obtained more than once via different paths. This "same-operation exception", as it is called by Snyder [Snyder 1986], has the advantage of having a simple, intuitive semantics, but it can lead to surprises if the underlying traits are changed. Suppose that trait TReadStream is re-implemented so that it no longer uses TPositionableStream but still supports the same behavior (*e.g.*, the method TPositionableStream>>atEnd is re-implemented in the trait TReadStream). This causes a conflict because trait ReadWriteStream now obtains two *different* methods atEnd. Thus, what may have appeared to be a strictly internal change to trait TReadStream becomes visible to one of its clients.

Although it may seem that this situation will lead to fragile hierarchies, we argue that it does not. When TReadStream re-implements atEnd, it is changing what it provides to its clients in a way that is less severe, but just as significant, as when it adds or removes methods. Any of these changes may introduce a naming conflict. But the resulting conflict is a purely *local* matter, that is, it can be corrected by the *direct clients* of TReadStream alone. ReadWriteStream can easily resolve the resulting conflict by excluding one atEnd or the other.

Let us examine two alternatives to our current rule. One alternative is for ReadWriteStream to "automatically" obtain either one atEnd or the other, as happens with linearly-ordered mixins. The problem with this is that the change to TReadStream would give the programmer no feedback, even though the semantics of ReadWriteStream might have changed.

The alternative suggested by Snyder is to abandon the "same-operation exception", and announce a conflict even if the same method is obtained multiple times [Snyder 1986]. In our example, this means that there would already be a conflict in the original scenario, and that the programmer would have to *arbitrarily* decide which of the two atEnd methods should be available in ReadWriteStream. We argue that this is more dangerous, because a later change to the atEnd provided by either TWriteStream or TReadStream will not be signalled as having a possible consequence on ReadWriteStream. With the current approach, the conflict is signalled at precisely the point in time at which it arises, which is when the programmer is able to adopt an informed resolution.

5.3 C++ Revisited

In the discussion above, we pointed out that traits emerged from the attempt to design a composition mechanism that combines the beneficial properties of both multiple inheritance and mixins. C++ is the only language we are aware of that allows the programmer to express both of these composition mechanisms: it has native support for multiple inheritance, and it also allows one to express mixins by using templates (see Section 3.2). This poses the interesting question whether it is possible to express a form of composition similar to traits in C++ by combining multiple inheritance with templates.

It turns out that this is indeed possible. The trick is that instead of expressing the reusable entities as generic classes and composing them into a linear inheritance hierarchy by template instantiation as suggested by VanHilst and Notkin [VanHilst and Notkin 1996a; 1996b] as well as by Smaragdakis and Batory [Smaragdakis and Batory 1998;

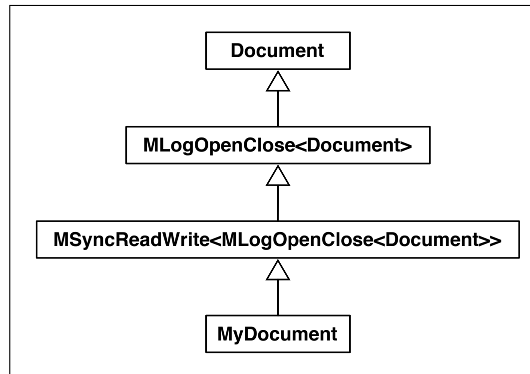


Fig. 19. Using C++ templates to simulate mixin composition

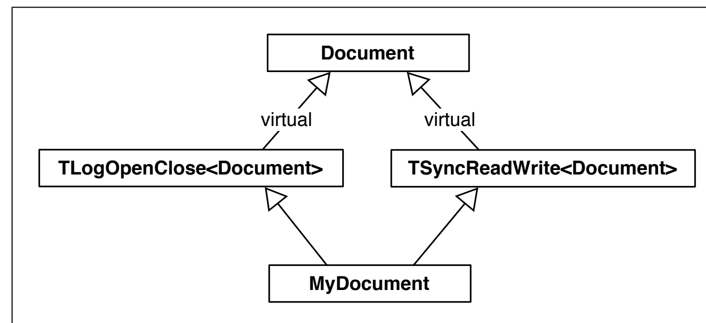


Fig. 20. Using C++ templates and virtual base classes to simulate trait composition

2000], we express them as classes with a *virtual* generic base class and then compose them into a *parallel* hierarchy using multiple inheritance.

The conceptual difference between these two approaches is illustrated in Figures 19 and 20. Figure 19 shows how the class MyDocument is derived from the class Document by a nested instantiation of the templates MLogOpenClose and MSyncReadWrite, which leads to a linear hierarchy (cf. Section 3.2). In contrast, Figure 20 shows how MyDocument is built from two templates TLogOpenClose and TSyncReadWrite, which are both applied to the class Document and are then composed using multiple inheritance.

The implementation of the template-based traits TSyncReadWrite and TLogOpenClose is shown in Figure 21. This figure does not show the method bodies because they are identical to the ones in the mixins MSyncReadWrite (Figure 7) and MLogOpenClose (Figure 8) discussed in Section 3.2. In fact, the only difference between the mixins and the corresponding traits is that the traits declare their generic base classes to be virtual.

Declaring the base class to be virtual is crucial as it would otherwise not be possible to correctly compose the traits using multiple inheritance. This is because composing these two traits means instantiating them with the *same* base class Document and then combining them using multiple inheritance. According to the semantics of virtual base classes [Stroustrup 1997], the resulting diamond situation has the key properties known from traits: the common base class Document is inherited only once, and methods in the

```
template <class Super>
class TLogOpenClose : virtual public Super {
    public:
    virtual void open() { ... };
    virtual void close() { ... };
    virtual void reset() { ... };

    protected:
    virtual void log(String s) { ... };
};

template <class Super>
class TSyncReadWrite : virtual public Super {
    public:
    virtual int read() { ... };
    virtual void write(int n) { ... };
    protected:
    virtual void acquireLock() { ... };
    virtual void releaseLock() { ... };
};

class MyDocument : public TLogOpenClose<Document>,
                   public TReadWriteSync<Document> {
    ...   // glue methods
};
```

Fig. 21. Implementing MyDocument as the composition of two "C++ traits"

traits TLogOpenClose and TSyncReadWrite override methods inherited from the common base class Document, while they are overridden by methods in the common subclass MyDocument. Furthermore, methods that are implemented by both traits TLogOpenClose and TSyncReadWrite result in a conflict that needs to be resolved in the subclass MyDocument.

C++ allows one to express composite traits by nesting the templates that represent traits. As an example, we can write a new trait TLogAndSync as a template class that is parameterized by super and inherits from the virtual base classes TLogOpenClose and TReadWriteSync, which are both instantiated with the new parameter super.

```
template <class Super>
class TLogAndSync : virtual public TLogOpenClose<Super>,
                    virtual public TReadWriteSync<Super> {};
```

A difference between traits and their C++ approximation is the fact that C++ supports only one of the three composition operators of traits: it can express trait sum ($+$) but not alias ($\rightarrow$) or exclusion ($-$). Whereas aliases can be simulated by disciplined use of the scope modifier ::, this is not the case for exclusion. This means that instead of excluding one or more conflicting methods from a composition, C++ requires the programmer to resolve every conflict by overriding the conflicting methods. While this may result in the same runtime behavior, it is not equivalent from a compositional point of view. When using exclusion, the introduction of a new conflicting method always leads to a conflict that requires explicit resolution, for example by excluding the new method. This is not the case for overriding in C++, where a newly occurring conflict is implicitly overridden by

the old conflict resolution code.

5.4 Traits as a General Composition Mechanism

Generalizing the above findings, we can say that C++ allows one to express trait-like composition by using a combination of nested templates and multiple inheritance with virtual base classes. Likewise, CLOS allows one to simulate traits using explicit linearization and meta programming. There are, however, enough differences between the traits model and such simulations to make traits a general composition mechanism in their own right.

We first observe that although C++ does not support the complete set of trait composition features, expressing traits in C++ can be achieved only by using a quite sophisticated combination of advanced language mechanisms such as nested templates and virtual base classes. As a consequence, using traits in C++ not only requires one to have a deep understanding of these mechanisms, but it also requires a lot of coding discipline to achieve the robustness benefits promised by the traits mechanism. As an example, the programmer has to avoid using nested scope modifiers (*e.g.*, Super::Super::reset()) to avoid fragility with respect to (distant) changes in the class and trait hierarchies. Similarly, one has to factor out all direct accesses to an overridden trait method into a *single* accessor method that is then called from all the other methods that require access to the overridden functionality. This avoids the fragility that arises if explicit calls to trait methods (*e.g.*, TColor::rgb()) are scattered throughout the source code of multiple methods.

The intrinsic complexity may be part of the reasons why this particular combination of C++ mechanisms was, to the best of our knowledge, not previously identified and suggested as a general composition idiom in C++. This is similar to template-based mixins in C++, which were scientifically investigated and described by VanHilst and Notkin [VanHilst and Notkin 1996b; 1996a] as well as by Smaragdakis and Batory [Smaragdakis and Batory 1998; 2000] only after mixins were proposed as a fundamental composition mechanism by Moon [Moon 1986] and later analyzed by Bracha and Cook [Bracha and Cook 1990]. As noted by VanHilst [VanHilst and Notkin 1996b], templates were previously used, for example in the C++ Standard Template Library (STL) [Musser and Saini 1996], for genericity (*i.e.*, writing data structures such as collections that can be used in the context of different types), but not for role composition using inheritance.

Another reason for the relevance of traits as a general composition mechanism is the fact that this form of composition cannot be expressed by any of the more recent object-oriented languages such as Java, C#, Python, and Ruby.

This is a strong indication that the C++ or CLOS approaches, which provide the programmer with an overwhelming array of mechanisms for feature composition, are not the best way to go. The main problem is that there is just too much of a risk that the average programmer will use and combine these mechanisms in an sub-optimal way, leading to code that is fragile, hard to understand and difficult to maintain.

In contrast, traits stand for a *single* composition mechanism that guarantees certain composition properties. By applying traits to a language such as Smalltalk, Java, or C#, we can therefore get most of the compositional power known from C++, but without its drawbacks.

6. IMPLEMENTATION

Traits as described in this paper are fully implemented in Squeak [Ingalls et al. 1997], an open-source dialect of Smalltalk. Our implementation consists of two parts: an extension of the Smalltalk language and an extension of the programming tools.

6.1 Language Extension

To add traits to Squeak, we extended the implementation of classes to include an additional instance variable to contain the information in the composition clause. This variable defines the traits used by the class, as well as 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 add to the method dictionary a second entry that associates the new name with the aliased method. Since compiled methods do not usually depend on the location at which 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. Methods that use the keyword super are an exception; these methods store an explicit reference to the superclass in their literal table. When a trait with such methods is applied to a class, these methods are copied into the class, and the literal that refers to the superclass is modified appropriately. This copying could be avoided by modifying the virtual machine to that it computes super when needed, rather than reading it from the literal table for the method.

In Smalltalk, classes are first-class objects; every class is an instance of a metaclass that defines the shape and the behavior of its singleton instance [Goldberg and Robson 1983]. In our implementation, we support this concept by introducing the notion of a *classtrait*; a classtrait can be associated with every trait. To preserve metaclass compatibility [Graube 1989; Bouraqadi-Saadani et al. 1998; Ducasse et al. 2005], whenever a trait is used in a class, the associated classtrait (if there is one) is automatically used in the metaclass. Consequently, a trait with an associated classtrait can be used only by classes, whereas a trait without a classtrait can be used by both classes and metaclasses.

When are classtraits necessary? Suppose that a trait T contains a method such as

```
T>>emptyCopy
    ↑ self class new: self size.
```

Clearly, this implies a requirement for a method size, that is, we know that whatever class eventually uses T must provide a method size. It also implies a *class* requirement for the method new:, that is, whatever class eventually uses T, its *metaclass* must provide a method new:. If the primitive implementation of new: in Behavior is not appropriate, the programmer might create a classtrait corresponding to T to contain a customized new: method. Whenever T is applied to a class, this (anonymous) classtrait will be automatically applied to the corresponding metaclass.

The treatment of classtraits by the traits browser is exactly parallel to the treatment of metaclasses. A classtrait can be created simply by switching from the "instance" side of the browser to the "class" side, and defining a method. Once a trait and a classtrait have been linked in this way, whenever the trait is used by a class, the corresponding classtrait will be used by the corresponding metaclass. No changes are necessary in the formal model to accommodate classtraits; classtraits are simply ordinary traits that happen to be designed for application to classes. Again, this is exactly parallel to the treatment of metaclasses in Smalltalk; metaclasses are simply ordinary objects that happen to be instances of the class Metaclass.

Because traits are simple and completely backwards 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 offsets of the instance variables in the object [Dixon et al. 1989].

Our implementation never duplicates source code, and duplicates byte code only if it includes sends to super. A program with traits therefore exhibits the same performance as the corresponding single inheritance program in which all the methods provided by traits are implemented directly in the classes that use those traits. This is especially remarkable because our implementation did not introduce any changes to the Squeak virtual machine. The only performance penalty results from the use of accessor methods, but such methods are in any case widely used because they improve maintainability. Modern JIT compilers routinely inline accessors, so we feel that requiring their use is now entirely justifiable.

6.2 Programming Tools

Besides introducing an extension to the language, our implementation also includes an extension of the programming tools, *i.e.*, the Smalltalk browser. We now give a brief overview of this extended browser; a more detailed description can be found in two companion papers [Schärli and Black 2003; Black and Schärli 2004].

For each class (and each trait), the browser shows the various traits from which it is composed. The flattening property means that the browser can flatten this hierarchical structure at any level, while preserving the semantics of the classes. 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 allow the programmer to work with different views of the code. On the one hand, the programmer can view and edit 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 or 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, which shows how the responsibilities of the class are decomposed into several traits, and how these traits are glued together to achieve the required behavior. This view is especially valuable because it allows a user to understand a class by seeing which traits it uses and examining the glue methods.

As in standard Smalltalk, the browser supports incremental compilation. Whenever a trait method is added, changed or deleted, all the users of that trait are immediately updated. The modifications are also analyzed to update 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 in generating the necessary glue code. Required methods that correspond to instance variable accessors are generated on request. Assistance is also provided in eliminating conflicts. The programmer is presented with a list of alternative implementations: choosing one of these implementations automatically generates the composition clause that excludes the others, and thus eliminates the conflict in favor of the chosen method.

7. EXPERIENCE

We first validated the usability of traits by using them to refactor the Smalltalk collection hierarchy, as implemented in version 3.2 of Squeak. In a second study, we used traits to solve metaclass composition problems and to refactor the Smalltalk kernel (as implemented in Squeak 3.6) by cleanly bootstrapping it with traits. In this section, we briefly summarize

the results of this work; interested readers are referred to two companion papers for more detail [Black et al. 2003; Ducasse et al. 2005].

7.1 Refactoring the Smalltalk Collection Hierarchy

The core classes of the Smalltalk 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 the properties of its instances, such as whether they are explicitly ordered (*e.g.*, Array), implicitly ordered (*e.g.*, SortedCollection), or unordered (*e.g.*, Set), whether they are extensible (*e.g.*, Bag), or immutable (*e.g.*, Interval), whether they are accessed by key (*e.g.*, Dictionary) or index (*e.g.*, OrderedCollection), and the operation used for element comparisons (*e.g.*, object identity, equality or a client-defined operator).

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 classes are forced to either duplicate code or to move methods higher in the hierarchy than makes conceptual sense, and then to disable these methods in the subclasses for which they are inappropriate [Cook 1992].

We solved these problems by creating traits for the different collection properties and combining them to build the required collection classes. 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 to achieve a sound hierarchy and avoid code duplication, we structured the code using fine-grained subtraits that allow us to reuse parts of the code outside of the collection hierarchy. For example, we introduced traits representing the behavior "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 composition of up to 22 traits, our tools take advantage of the flattening property to ensure that that this does not make the code any harder to understand than it was without traits. It is always possible to work with the hierarchy as if it were implemented with ordinary single-inheritance.

In total, we refactored 29 classes of the collection hierarchy, which originally implemented 635 methods. In the refactored versions, we built these classes from a total of 60 different traits. This allowed us to reduce the total number of methods to 567, which is just over 10% fewer methods than in the original implementation. In addition, the code for the trait-based implementation contains 12% less source code than the original. This is especially remarkable since nearly 9% of the methods in the original implementation were implemented "too high" in the hierarchy specifically to enable code sharing. With inheritance, the penalty for this is the repeated need to cancel inherited behavior (by using methods that cause a runtime error) in subclasses where it does not make sense[6]. In the

[6] In the original implementation, only 27% of the methods implemented "too high" were explicitly disabled in the subclasses where they did not belong. Therefore, eliminating such "error methods" accounts for only a small fraction of the methods we saved in our refactoring.

trait-based implementation, there is no need to resort to this tactic. This means that we were able to eliminate all cancellation of inherited methods from the refactored collection hierarchy.

Comparison to Mixins. It is clear that mixins could also be used to tackle the structural problems that we identified in the Smalltalk collection classes. We found, however, that because of the conceptual problems associated with mixins, achieving an equally fine-grained refactoring would be significantly more problematic with mixins than it is with traits.

Our refactored collection classes uses many traits, in one case as many as 22. Counting also the traits inherited from the superclasses, some our collection classes are built from a total of up to 35 traits (*e.g.*, OrderedCollection and Text). This is feasible because the sum operation lets us build a subclass from a group of traits in parallel, and the flattening property allows us to view and edit the nested trait structure at any level. In particular, the programmer can work with each class as if it were built without any traits at all.

In contrast, mixins must be applied one at a time, using the ordinary single inheritance operator. This would result in huge and hard to understand inheritance chains with up to 35 levels. This is especially problematic because mixin composition does not enjoy a flattening property. Whereas a class composed of a total of 35 traits can be consistently viewed and edited in a flat way (*i.e.*, as if only single inheritance were used), this is not possible with mixins, because mixins are composed using the inheritance operation, and thus the semantics of super-calls depends on the exact placement of the call in the inheritance chain.

In our refactored hierarchy, there are situations where several components are composed, but there is no ordering of mixins that would lead to the appropriate behavior. This is because there are sometimes *multiple* conflicts that need to be resolved by combining the conflicting methods or by excluding the methods that are not relevant. With traits, resolving these conflicts is simple because the composite entity can decide *independently* for each conflict how it should be resolved using exclusion and aliasing. Mixins, in contrast, need to be totally ordered and do not support exclusion and aliasing. Thus, the only way to resolve such conflicts would be to introduce additional glue mixins, which make such compositions more complex and harder to understand (see Section 3.1).

One example of this is the trait TSortedImpl where we had two conflicts: at:ifAbsent: and collect:, but no single subtrait that takes precedence for both of them. This is exactly the situation exclusion is designed for, and we obtained the desired behavior by excluding at:ifAbsent: from the subtrait TExtensibleSequencedImpl and collect: from TOrderedSortedCommonImpl.

Using mixins, the solution would be to either modify TExtensibleSequencedImpl, or to introduce a new intermediate mixin corresponding to TExtensibleSequencedImpl - {at:ifAbsent:}. Neither choice is desirable. Modifying the components is bad because it may break other places where these components are used. Introducing intermediate "glue mixins" makes the inheritance chains even longer and harder to understand (cf. Sections 2.2 and 3.1).

While composite mixins [Bracha 1992] allow the programmer to improve the understandability of such complex hierarchies by structuring the involved mixins, they do not solve the conceptual problems related to mixins (cf. Section 2.2). This is because composite mixins are based on the same linear form of composition as mixins, and they therefore suffer from the same problems such as fragility with respect to change and dispersal of

glue code. What would this mean in practice? Composite mixins would indeed allow us to structure the 21 mixins used to build SequencedImmutable into a handful of composite mixins. Nevertheless, the problems of fragility with respect to change, code dispersal, and so on, would then occur both *within* and *between* these composite mixins.

In the process of our refactoring work, we also encountered many situations where adding a new method to a component caused a conflict with another component in distant code. Thanks to the commutativity of trait sum and the requirement of explicit conflict resolution, all of these places were immediately detected, and we were able to re-establish the correct semantics by making an appropriate adjustment to the relevant trait composition clause. It was never necessary to modify other components, so we never found ourselves in a situation where resolving one conflict created two more.

With mixins, this would not have been the case. First, we would not have detected conflicting methods so easily because the order of the mixins *implicitly* "resolves" each conflict, although not necessarily in the way that the programmer intends! Second, even if we had noticed that a conflict had been resolved in an incorrect way, it would have been much harder to re-establish the correct behavior.

A comparison of our refactored collection classes to the mixin-based collection framework of Strongtalk [Bracha and Griswold 1993] provides more data on the effectiveness of mixins and traits. Both frameworks are based on Smalltalk-80 and are therefore quite comparable. Strongtalk has more collection classes, but uses only 10 different mixins, compared to 67 traits in our hierarchy. In particular, Strongtalk does not factor out characteristics such as extensible, implicitly sequenced, and explicitly sequenced; neither does it make aspects like enumeration reusable outside of the collection framework.

Of course, the fact that the designers of Strongtalk decided not to pursue a fine-grained decomposition into mixins does not mean that doing so would be impossible. But it is an indication that the designers of Strongtalk decided that the disadvantages of a finer structure outweighed the advantages. In contrast, we have found that with traits the fine-grained decomposition has only advantages.

Comparison to Multiple Inheritance. Multiple inheritance would also have solved many of the problems that we identified in the single inheritance version of the Smalltalk collection classes. Like mixins, however, multiple inheritance alone would not be expressive enough to achieve a fine-grained refactoring of the collection hierarchy.

For example, our refactored hierarchy uses several adaptor traits such as TIdentityAdaptor, which is used to turn collection classes like Set, WeakSet, and Dictionary into new classes that compare elements based on their identity rather than equality. But as described in Sections 2 and 3.2, multiple inheritance alone cannot express such adaptors without code duplication.

7.2 Applying Traits to Metaclasses and the Smalltalk Kernel

In class-based object-oriented programming, classes are used as instance generators and to implement the behavior of objects. In pure object-oriented languages such as CLOS and Smalltalk, classes themselves are first-class objects, being instances of so-called *metaclasses* [Ingalls 1976; Cointe 1987; Kiczales et al. 1991; Danforth and Forman 1994; Forman and Danforth 1999]. In the same way that classes (like String) define the properties of their instances (like 'Hello'), metaclasses (like String class) define the properties of their instances (String). Examples of class properties are *singleton*, *final*, and *abstract* [Ledoux

and Cointe 1996].

Because most of these class properties apply to many different classes, it is only natural to wish to share the corresponding code among multiple metaclasses. For example, both the class Number and the class Collection are abstract classes in Smalltalk, and it would be nice if this could be expressed by sharing the code that implements the *abstract* property among the corresponding metaclasses Number class and Collection class.

Unfortunately, giving the programmer explicit control over the metaclasses of a class leads to a variety of metaclass composition problems. These problems arise mainly because explicit metaclasses can break compatibility between the class and the metaclass level, *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 [Bouraqadi-Saadani et al. 1998; Bouraqadi 2004; Forman and Danforth 1999; Rivard 1997], but they always handle conflicting properties in an *ad-hoc* manner, alienating the meta-programmer. We have solved this problem in a *uniform* way by representing class properties as traits. This means that, like all other classes in the system, metaclasses are built as the composition of traits corresponding to the properties that they require; in the case of metaclasses, these are class properties such as singleton, abstract, and final.

While trait composition is used to add the required class properties to a metaclass, the inheritance hierarchy is used to ensure compatibility between the class and the metaclass level. This is achieved by retaining the parallel class and metaclass inheritance hierarchies found in Smalltalk: if a class A is the superclass of a class B, then the metaclass A class is the superclass of the metaclass B class.

As we have pointed out in our paper on metaclass composition [Ducasse et al. 2005], using the inheritance hierarchy for inter-level compatibility and traits for building metaclasses gives us the best of both worlds: it enables safe metaclass composition while retaining explicit control over what is being composed. Unlike other approaches, this approach is uniform because it uses the same object-oriented mechanisms, namely the complementary concepts of single inheritance and trait composition, to address design and implementation issues on both the base and the meta level. This has the advantage that the programmer does not have to learn a new composition mechanism that is exclusively targeted at composing metaclass properties.

Bootstrapping a new Smalltalk Kernel with Traits. Once traits were used to represent class properties, it was a natural extension to completely refactor the Squeak kernel by bootstrapping it with traits [Lienhard 2004]. Because the new kernel based on traits is an extension of the traditional Smalltalk kernel, it still contains the traditional Smalltalk kernel classes Behavior, ClassDescription, Metaclass, and Class [Goldberg and Robson 1983]. But unlike the classes in the old kernel, the new classes are built from various traits such as TInstantiator, TInstanceEnumerator, and TFamilyAccess, which correspond to the different responsibilities of the classes.

In addition, the new language kernel also contains the classes TraitBehavior, TraitDescription, Trait, and ClassTrait. These classes are necessary to represent traits, and they closely correspond to the traditional kernel classes. For example, just as Behavior provides basic functionality for compiling methods and managing a method dictionary in classes, TraitBehavior provides the same functionality for traits. These classes in the new kernel not only represent traits, but are also themselves built from traits. Examples include TMethod-

DictionaryManagement, TOrganization, and TCodeFileOut.

Besides improving the structure of the kernel, this is important because the new kernel classes that represent traits share many features with the traditional kernel classes that represent classes; putting the common code into traits enables these common features to be reused. This is especially true for the classes ClassDescription and TraitDescription, which share most of their methods.

Although the Smalltalk kernel is indeed small, the fact that we were able to cleanly bootstrap it with traits is another indication of the practical applicability of traits. For the language researcher, the refactored kernel has the advantage that experiments with the language are now much easier to cary out; this is because the different aspects of the language (*e.g.*, method dictionary management) are now available as traits that can be recomposed to create classes and objects with different properties.

8. RELATED WORK

In Sections 2 and 3, we have shown how multiple inheritance and various forms of mixin attempt to promote code reuse, and illustrated the problems that beset these generalizations of inheritance. In this section we compare traits to other approaches for structuring complex artifacts.

Other Reuse Constructs Called "Traits". Several other systems have used entities called "traits" to share and reuse implementation in ways that are related to the composition mechanism introduced in this paper.

One of these is the prototype-based language Self [Ungar and Smith 1987]. In Self, there is no notion of class; conceptually, each object defines its own format, methods, and inheritance relations. Objects are derived from other objects by cloning and modification. Objects can have one or more parent objects; messages that are not found in the object are looked for and delegated to a parent object. The order in which these parent objects are searched is not pre-defined[7], and it is an error for a selector to be found in more than one parent. In Self, explicit sends to such parent objects are called *resends*, of which there are two kinds: *directed resends* look in a specific parent object, while ordinary *resends* traverse all parent objects.

Self uses so-called *trait objects* to factor out common features [Ungar et al. 1991]. Similar to the notion of traits presented here, these trait objects are essentially groups of methods[8]. But unlike our traits, Self's trait objects do not support specific composition operators; instead, they are used as ordinary parent objects.

The software for the Xerox Star workstation was also implemented using entities called traits [Curry et al. 1982]. Traits were primitive entities used to build-up more complex objects. They were implemented as coding conventions in the Mesa programming language. This approach has more in common with other multiple inheritance approaches than with traits as presented in this paper. In particular, the Star traits differ from ours in their semantics for inheritance, their provision for conflict resolution capabilities, their ability to carry state, and their multiple implementations for a single method.

[7] In some older versions, Self featured sophisticated mechanisms to influence the search order. These mechanisms were later abandoned.

[8] Since Self is based around the notion of slots, which draws no difference between methods and data, nothing prevents a trait object from also containing state. But their goal is to just bundle methods.

The Larch Shared Language [Guttag et al. 1985] is also based on a construct called a trait; the relationship turns out to be more than name deep. Larch traits are fragments of specifications that can be freely reused at fine granularity. For example, it is possible to define a Larch trait such as IsEmpty that adds a single operation to an existing container datatype specification. But there are also significant differences between Larch traits and the traits presented in this work. In particular, adding a trait to a class is not intended to create a conservative extension of that class.

PIE. The Personal Information Environment (PIE) is a programming environment that supports the design, development, and documentation of Smalltalk programs [Goldstein and Bobrow 1980b; 1980a]. The PIE environment is based on a network of nodes that describe different types of entities — from small pieces such as a single procedure to much larger conceptual entities such as categories of classes or configurations of the system — in a uniform way. PIE features a wide variety of innovations such as context-sensitive descriptions (*i.e.*, properties with different associated values depending on the current context), meta-nodes containing a meta-description of the associated node, and a mechanism for unique identification of objects across an entire computing community [Bobrow and Goldstein 1980; Goldstein and Bobrow 1980b; 1980a].

In addition, PIE features a form of multiple inheritance based on a notion of multiple perspectives, which reflects the idea that a certain node may have different characteristics depending on the point of view from which it is considered. Hence, PIE allows the programmer to assign an arbitrary number of different perspectives (with independent superclasses) to a single node. In the initial version of PIE, the state of the object was represented entirely in the node, and the perspectives carried no state: they supplied method definitions only. Although this early form of perspectives bears a certain resemblance to traits, there are important conceptual differences. The most critical one is that unlike with traits, the methods provided by perspectives are not merged into the node where the perspectives are applied. This means that a node does not itself understand the messages implemented by its perspectives, and that a programmer therefore has to use a message pattern that explicitly states the class of the perspective providing the sent message.

While this design has the advantage that equally named methods of different perspectives never conflict, it also means that external clients depend on the structure of a node, and that perspectives are heavyweight entities that do not provide for fine-grained modularization of a node's methods. These conceptual differences between traits and perspectives are even more significant in the second and most recent version of PIE, where each perspective also carries its own state. Note that the notion of perspectives in PIE is based on very similar notions in FRL [Goldstein and Roberts 1977] and KRL [Bobrow and Winograd 1977], and that it is related to the approach employed by ThingLab [Borning 1981], a multiple inheritance constraint satisfaction system.

Template-based Approaches. There are several C++ template libraries such as the Standard Template Library (STL) [Musser and Saini 1996] and the Boost Lambda Library [Järvi et al. 2003], which implement a variety of parameterized data structures and functions such as collections and iterators. Whereas these parameterized data structures facilitate reuse because they are applicable in the context of different types, they are not directly related to the kind of feature composition that is the goal of traits.

Indeed, VanHilst and Notkin [VanHilst and Notkin 1996b] note that C++ templates can be used for two conceptually different kinds of parametrization: for *genericity* (*e.g.*, a

generic class Set with a parameterized element type) and for *composition* (*e.g.*, a class Color with a parameterized superclass)[9]. The conceptual difference between these two purposes of templates become especially apparent in a language like Smalltalk. Because Smalltalk is dynamically typed, templates are not necessary for writing the kind of generic data structures implemented in the STL, and because Smalltalk's built-in blocks are anonymous functions (*i.e.*, lambda abstractions), there is no need for the abstractions proposed by the Boost Lambda Library. But at the same time, Smalltalk lacks a flexible composition mechanism, which is why we extended the language with traits.

The RESOLVE discipline [Sitaraman and Weide 1994; Ogden et al. 1994; Edwards et al. 1994; Hollingsworth et al. 1994; Hollingsworth et al. 2000] for component-based software engineering is a set of software engineering design principles introduced by Hollingsworth in his doctoral dissertation [Hollingsworth 1992]. While the RESOLVE discipline is language independent, Sitaraman and Weide also developed specialized versions such as a version for C++ known as RESOLVE/C++ [Bucci et al. 1994]. Although the RESOLVE discipline covers many different kinds of software engineering principles such as avoiding aliasing problems by consistently using swapping and not assignment as the basic data movement mechanism, its main focus lies on component-based design, *i.e.*, the RESOLVE framework.

A key concept of the RESOLVE framework is the distinction between *abstract components* (specifications) and *concrete components* (implementations). The distinction between these two kinds of component allows each abstract component to be realized using any of several concrete components that correctly achieve the intended functionality but may for example differ in performance characteristics. RESOLVE components are also parameterized. In Resolve/C++, parametrization is achieved by making each component a C++ template with two kind of parameters: *conceptual parameters* for generic components such as a set that deals with elements of a parameterized type, and *realization parameters* that avoid concrete-to-concrete component coupling.

These realization parameters make the RESOLVE components similar to traits (and mixins), as they allow a programmer to apply and compose RESOLVE components in very flexible ways. Apart from this similarity, there are also significant differences between traits and RESOLVE. The traits mechanism is designed to be a simple and lightweight extension of single inheritance that enables one to build classes from a fine-grained composition of traits rather than an unstructured collection of individual methods. This is reflected by the fact that traits are implicitly parameterized and are composed in a way that is quite limited but in return guarantees certain properties, such as the flattening property, which are important for the understandability of such fine-grained structures.

In contrast, RESOLVE components are explicitly parameterized and are then composed and applied by (full or partial) instantiation rather than inheritance. Together with the distinction between abstract and concrete components, this makes the RESOLVE approach more complex and heavyweight — Edwards et al. describe the RESOLVE specification language as "rich and fairly complex" [Edwards et al. 1994] — but in return offers a wide variety of different kinds of compositions and conformance guarantees.

[9]Note that generics in most other languages such as C# and Java cannot express classes with parameterized superclasses. Therefore, they cannot be used to express mixin-like feature composition.

Mixin-related Approaches. GenVoca is a design methodology for creating application families and architecturally extensible software, *i.e.*, software that is customizable via module additions and removals [Batory and O'Malley 1992]. With GenVoca, class refinements are modeled as functions that take a program (*i.e.*, a GenVoca constant) as input and produce a feature-augmented program as output. While traits are purely behavioral, a GenVoca class refinement not only can introduce or override methods, but also can add new data members and constructors to a target class. A more fundamental difference between traits and GenVoca is that traits rely on the composition operators to guarantee important properties for making the resulting classes easy to understand (*e.g.*, the flattening property) and robust with respect to change (*e.g.*, commutativity and explicit conflict resolution). In contrast, GenVoca's main innovations are the layering and scaling of refinements that allow one to generate high-performance systems for a target domain. Indeed, the actual implementation of GenVoca refinements is based on existing mechanisms such as mixins [Batory et al. 2003].

Mixin layers [Smaragdakis and Batory 1998] are a technique for implementing layered object-oriented designs (*e.g.*, collaboration-based designs). Mixin layers are similar to mixins but scaled to the granularity of multiple-classes. Mixin layers address the scalability problems that can appear in role-based designs [VanHilst and Notkin 1996a; 1996b], but they still suffer from the fragility problems we identified for mixins (such as fragility with respect to change) because they are based on mixins as the fundamental composition mechanism.

Smaragdakis also showed how one can develop layered software using common Unix (Linux and Solaris) dynamic libraries [Smaragdakis 2002]. The idea is that, from an object-oriented design standpoint, dynamic libraries are analogous to components in a mixin-based object system. This enables one to use libraries in a layered fashion, mixing and matching different libraries, while ensuring that the result remains consistent. As with all the other mixin-based approaches, composition is linear and the composition order is crucial to the semantics of the composition. As a consequence, this form of dynamic library composition also suffers from the mixin-related problems that we have addressed with our work on traits (cf. Sections 2 and 3)

Aspect-oriented Programming (AOP). Aspect-oriented programming [Kiczales et al. 1997] allows the programmer to encapsulate concerns that cross-cut class boundaries in a construct called an aspect. Both aspects and traits can add new methods to existing classes. Aspects can also weave code before or after the execution of a method, an effect traits achieve using method overriding and explicit calls to super. In addition, most implementations of aspect-oriented programming such as AspectJ [Kiczales et al. 2001] support weaving code at more fine-grained join points such as field accesses, which is not supported by traits.

Despite the fact that traits and aspects can be used for similar purposes, there are fundamental differences between the two approaches. By definition, aspects are concerns that cannot be cleanly encapsulated in a generalized procedure (*i.e.*, object, method, mixin). This means that in contrast to traits, aspects are neither designed nor used to build classes and components from scratch, but rather to alter the performance or semantics of the components in systemic ways. Thus, a single aspect can be designed to modify the behavior of methods spread across the object-oriented decomposition, *i.e.*, in many classes. A trait cannot do this.

Other Modularity and Composition Models. Delegation (also known as "object-based inheritance") is another form of composition that side-steps many of the problems related to class-based inheritance [Kniesel 1999]. In contrast to traits, delegation is designed to support *dynamic* component adaptation.

The Jigsaw modularity framework, developed by Bracha in his doctoral dissertation [Bracha 1992], defines module composition operators such as merge, rename and restrict that are strikingly similar to our trait sum, alias and exclusion operators. For example, Bracha's merge, like our sum operator, is commutative. Although there are differences in the details of the definitions (for example, in how conflicts are handled), the more significant differences are in motivation and setting. Jigsaw is intended as a complete framework for module manipulation in the large, and makes assumptions appropriate to that setting: namespaces, declared types and requirements, full renaming, and semantically meaningful nesting. Traits are intended to supplement existing languages by promoting reuse in the small, and consequently do not define namespaces, do not declare types, infer their requirements, do not allow renaming, and do not give a meaning to nesting. The Jigsaw operation set also aims for completeness, whereas in the design of traits we explicitly gave up completeness for simplicity. Nevertheless, the similarity of the core operation sets is encouraging, given that they were defined independently.

Logtalk [Moura 2003] is an open source object-oriented extension to the Prolog programming language. It supports both prototypes and classes. In addition, it supports component-based programming using a mechanism called categories that is designed to share code between classes. Despite a superficial resemblance between Logtalk categories and traits, there are many differences between the two mechanisms. Logtalk does not support aliasing or exclusion, but uses a depth-first lookup to *implicitly* resolve any conflicts, and it suffers from scalability problems as categories cannot be composed from other categories.

Mohnen proposes an extension of Java that allows interfaces to have default implementations [Mohnen 2002]. As such, classes that implements such an interface can explicitly state that they want to use the default implementation offered by that interface (if any). If more than one interface mentions the same method, a method body must be provided. The system is implemented as a pure compiler-extension for Java. Conflicts are flagged automatically but require the developer to resolve them manually. The composition mechanism lacks exclusion and aliasing.

Caesar's collaboration interfaces are similar to traits in that they include the declaration of *expected* methods, *i.e.*, those that classes must provide when bound to an interface [Mezini and Ostermann 2002]. Thus, Caesar's interface concept can simulate traits by binding an interface to a class and then combining it with a specific implementation. However, Caesar has no special compositional construct for dealing with conflicts. Instead, Caesar is designed to use one of the conflict resolution strategies known from multiple inheritance languages such as C++, leading to problems similar to those described in Sections 2 and 3. Moreover, Caesar is based on explicit wrappers, which can be costly at runtime, while the semantics of traits is compatible with single inheritance and does not cause a run-time penalty.

Mezini also proposed an approach to behavior composition in a class-based environment that is based on the encapsulated object model of class-based inheritance, but introduces an explicit combination layer between objects and classes [Mezini 1997]. The definition of the behavior of an evolving object is divided between a class that provides the standard

behavior of the object and a set of mixin-like software modules, called adjustments. One of the main differences from traits is that Mezini's approach is more dynamic and complex. In fact, a combiner-metaobject is associated with each evolving object, and is responsible for the compositional aspects of the object's behavior. This means that the combiner-metaobject uses the adjustments to define the environment where the messages sent to the object are evaluated.

9. CONCLUSIONS AND FUTURE WORK

This paper has introduced traits as a mechanism for building and structuring the classes in object-oriented programs. The same trait can be reused in many classes, irrespective of the position of those classes in the inheritance hierarchy. Traits can be manipulated using a set of operators—sum, overriding, 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 offer an ideal extension to single inheritance languages. Traits are completely backwards compatible with Smalltalk and do not require one to modify or extend the method syntax of the underlying language. Furthermore, the flattening property guarantees that the resulting code is no less understandable than the original, because it is always possible to both view and edit the code as if it were written using single inheritance without traits.

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 have used traits successfully in several case studies such as the refactoring of the Smalltalk collection classes. This is a strong indication of the usability of traits for realistic and non-trivial problems. This refactoring also showed that traits are suitable for modularizing classes that have already been built, and that traits raise the level of abstraction when building new classes. As we worked with the refactored hierarchy, we were impressed with the power of the flattening property, which made understanding classes that are built from composite traits quite a simple matter.

Based on our experience, both from designing traits and from using them together with other programmers in the context of our case studies, we also developed a methodology for programming with traits [Black and Schärli 2004]. We found that the conceptual difference between traits and classes leads to a natural distinction regarding how and when these two concepts should be used. In short, the practical role of traits is to capture the different variants of individual protocols; these variants can then be composed to build composite protocols and finally classes. This frees the class hierarchy to be used for the conceptual classification of objects.

We are currently working to integrate traits into Java, and on a project initiated by Microsoft for porting traits to the .NET platform. In both of these languages the type system makes the simple approach that works so well for Smalltalk more problematic. For example, the type of a method in a trait may depend on the class in which it is eventually used, and the rules for method overloading have to be taken into account. However, given the experience from our work so far, and the availability of traits in the statically-typed language Scala [Odersky et al. 2004], we are convinced that these problems can be overcome.

As future work we would like to (1) evaluate the impact of the introduction of namespaces and encapsulation on the flattening property, (2) consider the effects of allowing traits to specify state variables, (3) extend trait composition so that it can replace inheritance, (4) evaluate the possibility of using traits to modify the behavior of individual instances at run-time, and (5) further explore the application of traits to the refactoring of complex class hierarchies.

Acknowledgments

We would like to thank Paolo Bonzini, Gilad Bracha, William Cook, Erik Ernst, Robert Hirschfeld, Andreas Raab and Jean-Guy Schneider for their rich interaction and valuable comments while developing traits and writing this paper. We thank Mira Mezini and Klaus Ostermann for the discussions about Caesar. 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, Oct. 2002 - Sept. 2004) and "A Unified Approach to Composition and Extensibility" (SNF Project No. 200020-105091/1, Oct. 2004 - Sept. 2006). This work was also supported by the National Science Foundation of the USA under awards CCR-0098323, CCR-0313401, and CCF-0520346.

REFERENCES

AMERICA, P. AND VAN DER LINDEN, F. 1990. A parallel object-oriented language with inheritance and subtyping. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*. Vol. 25. 161–168.

ANCONA, D., LAGORIO, G., AND ZUCCA, E. 2000. Jam — a smooth extension of Java with mixins. In *ECOOP 2000*. Number 1850 in Lecture Notes in Computer Science. 145–178.

BAK, L., BRACHA, G., GRARUP, S., GRIESEMER, R., GRISWOLD, D., AND HÖLZLE, U. 2002. Mixins in Strongtalk. In *ECOOP '02 Workshop on Inheritance*.

BARRETT, K., CASSELS, B., HAAHR, P., MOON, D. A., PLAYFORD, K., AND WITHINGTON, P. T. 1996. A monotonic superclass linearization for dylan. In *Proceedings OOPSLA '96, ACM SIGPLAN Notices*. 69–82.

BATORY, D. AND O'MALLEY, S. 1992. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*.

BATORY, D., SARVELA, J. N., AND RAUSCHMAYER, A. 2003. Scaling step-wise refinement. In *Proceedings of the 25th international conference on Software engineering*. IEEE Computer Society, 187–197.

BLACK, A. P. AND SCHÄRLI, N. 2004. Traits: Tools and methodology. In *Proceedings ICSE 2004*. 676–686.

BLACK, A. P., SCHÄRLI, N., AND DUCASSE, S. 2003. Applying traits to the Smalltalk collection hierarchy. In *Proceedings OOPSLA'03 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*. Vol. 38. 47–64.

BOBROW, D. G. AND GOLDSTEIN, I. P. 1980. Representing design alternatives. In *Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior*.

BOBROW, D. G. AND WINOGRAD, T. 1977. An overview of KRL, a knowledge representation language. *Cognitive Science 1,* 1, 3–46.

BORNING, A. 1981. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM TOPLAS 3,* 4 (Oct.), 353–387.

BOURAQADI, N. 2004. Safe metaclass composition using mixin-based inheritance. *Journal of Computer Languages, Systems and Structures 30,* 1-2 (Apr.), 49–61.

BOURAQADI-SAADANI, N. M. N., LEDOUX, T., AND RIVARD, F. 1998. Safe metaclass programming. In *Proceedings OOPSLA '98*. 84–96.

BRACHA, G. 1992. The programming language Jigsaw: Mixins, modularity and multiple inheritance. Ph.D. thesis, Dept. of Computer Science, University of Utah.

BRACHA, G. AND COOK, W. 1990. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*. Vol. 25. 303–311.

BRACHA, G. AND GRISWOLD, D. 1993. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*. Vol. 28. 215–230.

BRUCE, K. B., CARDELLI, L., CASTAGNA, G., GROUP, T. H. O., LEAVENS, G. T., AND PIERCE, B. 1995. On binary methods. *Theory and Practice of Object Systems 1,* 3, 221–242.

BUCCI, P., HOLLINGSWORTH, J. E., KRONE, J., AND WEIDE, B. W. 1994. Part iii: implementing components in RESOLVE. *SIGSOFT Softw. Eng. Notes 19,* 4, 40–51.

CANNON, H. I. 1982. Flavors: A non-hierarchical approach to object-oriented programming. Tech. rep., Symbolics Inc.

CASTAGNA, G. 1995. Covariance and contravariance: conflict without a cause. *ACM Trans. Program. Lang. Syst. 17,* 3, 431–447.

COINTE, P. 1987. Metaclasses are first class: the ObjVlisp model. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*. Vol. 22. 156–167.

COOK, S. 1987. OOPSLA '87 Panel P2: Varieties of inheritance. In *OOPSLA '87 Addendum To The Proceedings*. ACM Press, 35–40.

COOK, W., HILL, W., AND CANNING, P. 1990. Inheritance is not subtyping. In *Proceedings POPL '90*. San Francisco.

COOK, W. AND PALSBERG, J. 1989. A denotational semantics of inheritance and its correctness. In *Proceedings OOPSLA '89*. Vol. 24. 433–443.

COOK, W. R. 1992. Interfaces and Specifications for the Smalltalk-80 Collection Classes. In *Proceedings of OOPSLA '92 (7th Conference on Object-Oriented Programming Systems, Languages and Applications)*. Vol. 27. ACM Press, 1–15.

CURRY, G., BAER, L., LIPKIE, D., AND LEE, B. 1982. TRAITS: an approach to multiple inheritance subclassing. In *Proceedings ACM SIGOA, Newsletter*. Vol. 3. Philadelphia.

DANFORTH, S. AND FORMAN, I. R. 1994. Derived metaclass in SOM. In *Proceedings of TOOLS EUROPE '94*. 63–73.

DIXON, R., MCKEE, T., VAUGHAN, M., AND SCHWEIZER, P. 1989. A fast method dispatcher for compiled languages with multiple inheritance. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*. Vol. 24. 211–214.

DUCASSE, S., SCHÄRLI, N., AND WUYTS, R. 2005. Uniform and safe metaclass composition. *Journal of Computer Languages, Systems and Structures 31,* 3-4 (Dec.), 143–164.

DUCOURNAU, R., HABIB, M., HUCHARD, M., AND MUGNIER, M. 1992. Monotonic conflict resolution mechanisms for inheritance. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*. Vol. 27. 16–24.

DUGGAN, D. AND TECHAUBOL, C.-C. 2001. Modular mixin-based inheritance for application frameworks. In *Proceedings OOPSLA 2001*. 223–240.

EDWARDS, S. H., HEYM, W. D., LONG, T. J., SITARAMAN, M., AND WEIDE, B. W. 1994. Part ii: specifying components in RESOLVE. *SIGSOFT Softw. Eng. Notes 19,* 4, 29–39.

FLATT, M., KRISHNAMURTHI, S., AND FELLEISEN, M. 1998. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 171–183.

FORMAN, I. R. AND DANFORTH, S. 1999. *Putting Metaclasses to Work: A New Dimension in Object-Oriented Programming*. Addison-Wesley.

GOLDBERG, A. AND ROBSON, D. 1983. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass.

GOLDSTEIN, I. P. AND BOBROW, D. G. 1980a. Descriptions for a programming environment. In *Proceedings of the First Annual Conference of the National Association for Artificial Intelligence*.

GOLDSTEIN, I. P. AND BOBROW, D. G. 1980b. Extending object-oriented programming in Smalltalk. In *Proceedings of the Lisp Conference*. 75–81.

GOLDSTEIN, I. P. AND ROBERTS, R. B. 1977. Nudge, a knowledge-based scheduling program. In *Proceedings of the Fifth International Joint Conference on Artifical Intelligence*. 257–263.

GRAUBE, N. 1989. Metaclass compatibility. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*. Vol. 24. 305–316.

GUTTAG, J. V., HORNING, J. J., AND WING, J. M. 1985. The Larch family of specification languages. *IEEE Transactions on Software Engineering 2,* 5 (Sept.), 24–36.

HARRISON, W. AND OSSHER, H. 1993. Subject-oriented programming (a critique of pure objects). In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*. Vol. 28. 411–428.

HOLLINGSWORTH, J. E. 1992. Software component design-for-reuse: A language independent discipline applied to ada. Ph.D. thesis, Dept. of Computer & Information Science, The Ohio State University, Columbus, OH.

HOLLINGSWORTH, J. E., BLANKENSHIP, L., AND WEIDE, B. W. 2000. Experience report: Using RESOLVE/C++ for commercial software. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press, 11–19.

HOLLINGSWORTH, J. E., SREERAMA, S., WEIDE, B. W., AND ZHUPANOV, S. 1994. Part iv: RESOLVE components in Ada and C++. *SIGSOFT Softw. Eng. Notes 19,* 4, 52–63.

INGALLS, D. 1976. The Smalltalk-76 programming system design and implementation. In *POPL'76, Principles of Programming Languages*. ACM Press, 9–16.

INGALLS, D., KAEHLER, T., MALONEY, J., WALLACE, S., AND KAY, A. 1997. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*. ACM Press, 318–326.

JÄRVI, J., POWELL, G., AND LUMSDAINE, A. 2003. The lambda library: unnamed functions in C++. *Softw. Pract. Exper. 33,* 3, 259–291.

KEENE, S. E. 1989. *Object-Oriented Programming in Common-Lisp*. Addison Wesley.

KICZALES, G., DES RIVIÈRES, J., AND BOBROW, D. G. 1991. *The Art of the Metaobject Protocol*. MIT Press.

KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. 2001. An overview of AspectJ. In *Proceeding ECOOP 2001*. Number 2072 in LNCS. Springer Verlag, 327–353.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-Oriented Programming. In *Proceedings ECOOP '97*, M. Aksit and S. Matsuoka, Eds. LNCS, vol. 1241. Springer-Verlag, Jyvaskyla, Finland, 220–242.

KNIESEL, G. 1999. Type-safe delegation for run-time component adaptation. In *Proceedings ECOOP '99*, R. Guerraoui, Ed. LNCS, vol. 1628. Springer-Verlag, Lisbon, Portugal, 351–366.

LALONDE, W. AND PUGH, J. 1991. Subclassing $\neq$ Subtyping $\neq$ Is-a. *Journal of Object-Oriented Programming 3,* 5 (Jan.), 57–62.

LALONDE, W. R. 1989. Designing families of data types using exemplars. *Transactions on Programming Languages and Systems 11,* 2 (Apr.), 212–248.

LEDOUX, T. AND COINTE, P. 1996. Explicit metaclasses as a tool for improving the design of class libraries. In *Proceedings of ISOTAS '96, LNCS 1049*. JSSST-JAIST, 38–55.

LIENHARD, A. 2004. Bootstrapping Traits. M.S. thesis, University of Bern.

MADSEN, O. L., MAGNUSSON, B., AND MOLLER-PEDERSEN, B. 1990. Strong typing of object-oriented languages revisited. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*. Vol. 25. 140–150.

MENS, T. AND VAN LIMBERGHEN, M. 1996. Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object Oriented Systems 3,* 1, 1–30.

MEYER, B. 1988. *Object-oriented Software Construction*. Prentice-Hall.

MEYER, B. 1992. *Eiffel: The Language*. Prentice-Hall.

MEYER, B. 1997. *Object-Oriented Software Construction*, Second ed. Prentice-Hall.

MEZINI, M. 1997. Dynamic object evolution without name collisions. In *Proceedings ECOOP '97*. Springer-Verlag, 190–219.

MEZINI, M. AND OSTERMANN, K. 2002. Integrating independent components with on-demand remodularization. In *Proceedings OOPSLA 2002*. 52–67.

MOHNEN, M. 2002. Interfaces with default implementations in Java. In *Conference on the Principles and Practice of Programming in Java*. ACM Press, Dublin, Ireland, 35–40.

MOON, D. A. 1986. Object-oriented programming with Flavors. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*. Vol. 21. 1–8.

MOURA, P. 2003. Logtalk. Ph.D. thesis, Universidade da Beira Interior.

MUSSER, D. R. AND SAINI, A. 1996. *STL Tutorial and Reference Guide*. Addison Wesley.

ODERSKY, M., ALTHERR, P., CREMET, V., EMIR, B., MANETH, S., MICHELOUD, S., MIHAYLOV, N., SCHINZ, M., STENMAN, E., AND ZENGER, M. 2004. An overview of the Scala programming language. Technical Report 64, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland.

OGDEN, W. F., SITARAMAN, M., WEIDE, B. W., AND ZWEBEN, S. H. 1994. Part i: the RESOLVE framework and discipline: a research synopsis. *SIGSOFT Softw. Eng. Notes 19,* 4, 23–28.

RIEL, A. 1996. *Object-Oriented Design Heuristics*. Addison Wesley, Boston MA.

RIVARD, F. 1997. Évolution du comportement des objets dans les langages à classes réflexifs. Ph.D. thesis, Ecole des Mines de Nantes, Université de Nantes, France.

SAKKINEN, M. 1989. Disciplined inheritance. In *Proceedings ECOOP '89*, S. Cook, Ed. Cambridge University Press, Nottingham, 39–56.

SAKKINEN, M. 1992. The darker side of C++ revisited. *Structured Programming 13,* 4, 155–177.

SCHÄRLI, N. 2005. Traits — composing classes from behavioral building blocks. Ph.D. thesis, University of Berne.

SCHÄRLI, N. AND BLACK, A. P. 2003. A browser for incremental programming. Technical Report CSE-03-008, OGI School of Science & Engineering, Beaverton, Oregon, USA. Apr.

SCHÄRLI, N., DUCASSE, S., NIERSTRASZ, O., AND BLACK, A. 2003. Traits: Composable units of behavior. In *Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming)*. LNCS, vol. 2743. Springer Verlag, 248–274.

SCHÄRLI, N., DUCASSE, S., NIERSTRASZ, O., AND WUYTS, R. 2004. Composable encapsulation policies. In *Proceedings ECOOP 2004 (European Conference on Object-Oriented Programming)*. LNCS 3086. Springer Verlag, 26–50.

SITARAMAN, M. AND WEIDE, B. 1994. Component-based software using RESOLVE. *SIGSOFT Softw. Eng. Notes 19,* 4, 21–22.

SMARAGDAKIS, Y. 2002. Layered development with (Unix) dynamic libraries. In *Proceedings ICSR 2002*, C. Gacek, Ed. Lecture Notes in Computer Science, vol. 2319. Springer, 33–45.

SMARAGDAKIS, Y. AND BATORY, D. 1998. Implementing layered design with mixin layers. In *Proceedings ECOOP '98*, E. Jul, Ed. LNCS, vol. 1445. Brussels, Belgium, 550–570.

SMARAGDAKIS, Y. AND BATORY, D. 2000. Mixin-based programming in C++. In *2nd Symposium on Generative and Component-Based Software Engineering (GCSE 2000)*. Erfurt, Germany.

SNYDER, A. 1986. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*. Vol. 21. 38–45.

STEELE, G. L. 1990. *Common Lisp The Language*, Second ed. Digital Press.

STEFIK, M. AND BOBROW, D. G. 1985. Object-oriented programming: Themes and variations. *The AI Magazine*.

STROUSTRUP, B. 1986. *The C++ Programming Language*. Addison Wesley, Reading, Mass.

STROUSTRUP, B. 1997. *The C++ Programming Language*, Third ed. Addison Wesley.

SWEENEY, P. F. AND GIL, J. Y. 1999. Space and time-efficient memory layout for multiple inheritance. In *Proceedings OOPSLA '99*. ACM Press, 256–275.

TAIVALSAARI, A. 1996. On the notion of inheritance. *ACM Computing Surveys 28,* 3 (Sept.), 438–479.

TARR, P., OSSHER, H., HARRISON, W., AND SUTTON, JR, S. M. 1999. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *Proceedings of ICSE '99*. Los Angeles CA, USA, 107–119.

UNGAR, D., CHAMBERS, C., CHANG, B.-W., AND HÖLZLE, U. 1991. Organizing programs without classes. *LISP and SYMBOLIC COMPUTATION: An international journal 4,* 3.

UNGAR, D. AND SMITH, R. B. 1987. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*. Vol. 22. 227–242.

VANHILST, M. AND NOTKIN, D. 1996a. Using C++ Templates to Implement Role-Based Designs. In *JSSST International Symposium on Object Technologies for Advanced Software*. Springer Verlag, 22–37.

VANHILST, M. AND NOTKIN, D. 1996b. Using Role Components to Implement Collaboration-Based Designs. In *Proceedings OOPSLA '96*. ACM Press, 359–369.

WEGNER, P. AND ZDONIK, S. B. 1988. Inheritance as an incremental modification mechanism or what like is and isn't like. In *Proceedings ECOOP '88*, S. Gjessing and K. Nygaard, Eds. LNCS, vol. 322. Springer-Verlag, Oslo, 55–77.