Traits: Tools and Methodology[†]

Andrew P. Black
OGI School of Science & Engineering,
Oregon Health and Science University, USA
black@cse.ogi.edu

Nathanael Schärli
Software Composition Group,
University of Bern, Switzerland
schaerli@iam.unibe.ch

Abstract

Traits are an object-oriented programming language construct that allow groups of methods to be named and reused in arbitrary places in an inheritance hierarchy. Classes can use methods from traits as well as defining their own methods and instance variables. Traits thus enable a new style of programming, in which traits rather than classes are the primary unit of reuse. However, the additional sub-structure provided by traits is always optional*: a class written using traits can also be viewed as a flat collection of methods, with no change in its semantics.*

This paper describes the tool that supports these two alternate views of a class, called the traits browser, and the programming methodology that we are starting to develop around the use of traits.

1. Introduction

The ability to reuse existing code is one of the compelling strengths of the object-oriented paradigm. Of course, procedural and functional programming also provide for reuse, but only in the most limited way: a whole procedure or function can be called, but it cannot be modified. Inheritance improves on procedural reuse in two ways. First, it provides for the reuse not just of a single procedure but of a group of related procedures, called methods, and the state on which they depend. Second, inheritance allows incremental modification of the reused component: instead of requiring clairvoyance on the part of the original programmer in anticipating every variation in functionality that the re-user might require, inheritance allows the re-user to selectively override parts of the original behaviour.

Because of these advantages, inheritance-based languages have become dominant. Yet single inheritance is widely recognized as deficient when it comes to reusing behaviour in multiple places in an inheritance hierarchy. Multiple inheritance and mixins have been proposed as solutions to this problem, but a common perception in the language design community is that these extensions introduce more problems than they solve.

We have previously argued that traits remedy the deficiencies of single inheritance without introducing the problems that beset multiple inheritance and mixins [10]. By refactoring the Smalltalk collection classes, we have shown that a mature code base can benefit from traits: the refactored classes contained less code, and yet exhibited more uniform interfaces [2]. These experiences with traits have also caused us to realize that support from appropriate tools is critical, as is a programming methodology that leverages traits' theoretical properties and the strengths of the tools.

The contributions of this paper are a description of the traits browser (section 4), the tool that we have developed to support programming with traits, and of the programming methodology that is evolving around it (section 5). We have attempted to make the paper self-contained by summarizing what traits are (section 3) and the problem that they solve (section 2); these sections condense more detailed descriptions that have appeared previously [10].

Although the current implementation of traits is in Smalltalk, we believe that traits can be applied with equal benefit to other single inheritance languages such as Java, and so we have tried to make this paper accessible to those familiar with object-oriented concepts but unfamiliar with Smalltalk. Related work is discussed where it is relevant; previous papers contain a detailed discussion of the relationship of traits to mixins and multiple inheritance [10], and a comparison of the traits browser to other browsers [9].

2. What is the Problem?

Edsger Dijkstra has written eloquently of "our inability to do much" [5]. One of the ways in which object-oriented programming helps us to do more, to cope with the ever-increasing variety of objects that our programs are asked to

† This research was partially supported by the National Science Foundation of the United States under awards CDA-9703218, CCR-0098323 and CCR-0313401, and by Swiss National Foundation project 2000-067855.02.

manipulate, is by encouraging the programmer to provide diverse objects with uniform protocol.

The notion of protocol, also known as interface, is crucial in all object-oriented programs, whether or not the language in which they are written has a syntactic construct to capture it. Uniformity of protocol is encouraged by inheritance, because by default the protocol of a subclass will be a superset of the protocol of its superclass. But classes that are not related by inheritance should also, very often, share the same protocol. Java's **interface** and **implements** constructs allow a programmer to state that two classes *should* share a protocol, but they do nothing to help translate that desire into code.

To see the value of uniform protocol, consider the enumeration protocol in Smalltalk. This protocol is part of the interface of Collection and its subclasses, and consists of the following messages. [1]

allSatisfy:	anySatisfy:	associationsDo:
collect:	collect:thenSelect:	count:
detect:	detect:ifNone:	detectMax:
detectMin:	detectSum:	difference:
do:	do:separatedBy:	do:without:
groupBy:having:	inject:into:	intersection:
noneSatisfy:	reject:	select:
select:thenCollect:	union:	

These messages implement various internal iterators over the target collection. For example, select: takes a boolean block (a predicate) as argument and returns a new collection containing those elements of the target collection for which the predicate yields true. All of the messages in the enumeration protocol are understood by all of the classes of collection; this makes it trivial to write code that is robust to changes in the specific kind of collection that is eventually provided.

Now consider the class Path, which represents an ordered sequence of points: arcs, curves, lines and splines are all implemented as subclasses of Path. The class Path is itself a subclass of DisplayObject, and thus not able to inherit from Collection. Consequently, although Path implements some of the more basic kinds of collection-like behaviour — for example, it has methods for select: and collect: — it does *not* implement the full enumeration protocol.

We first became aware of this deficiency when preparing a tutorial on Squeak, a dialect of Smalltalk [7]. The code

```
p := Path fromUser.
r := Rectangle fromUser.
pc := p select: [ :each | r containsPoint: each ].
pc displayOn: Display.
```

asks the user to input a series of points that defines a Path p on the display, and to then define a Rectangle r. It then creates a new Path pc that contains only those points from p that are inside r. But this code doesn't work: it produces a "message not understood" error, because p, although conceptually a collection of points, is actually a Path and thus does not understand the select: message.

Although the user of a Path can program around this deficiency, that ought not to be the user's responsibility! Instead, the *implementor* of Path should ensure that it understands the entire collection protocol. But this is an awesome task: the existing implementation in class Collection cannot be reused by inheritance, so many methods would need to be duplicated. In addition to the score of methods missing from the enumeration protocol, there are a dozen other protocols that must also be implemented if Path is to behave as a collection.

We have found similar problems in many places in Squeak. Although isEmpty is defined in 21 classes, only two of them also define notEmpty, and only one also defines ifEmpty: and ifNotEmpty: . Thus, the client of these classes cannot program to a uniform interface, and is subjected to the unnecessary burden of keeping track of exactly which messages the target object understands. The example of RectangleMorph not implementing all of the protocol understood by Rectangle objects is discussed at length in a previous paper [2], where we make the point that the reason for this non-uniformity in protocol is not bad programming, but bad technology. With single inheritance, the only way of making the protocols uniform is wholesale code duplication, which is probably a greater evil than non-uniformity.

Traits provide a simple solution to this dilemma that avoids duplication of both source and compiled code and also improves modularity, thus making the classes concerned easier to understand.

3. What are Traits?

A trait is a first-class collection of named methods — an implementation of a protocol. The purpose of a trait is to make that protocol implementation reusable in whatever classes need it. For simplicity, we make the restriction that the methods must be "pure behaviour," that is, they cannot directly reference any instance variables. We will now describe traits in some detail; readers already familiar with traits can safely omit this section.

Traits fulfill their purpose by being composed into other traits and eventually into classes. A trait has no superclass; the keyword **super** can appear in a trait method, but it remains unbound until the trait is eventually used in a class.

The power and simplicity of traits comes from the composition operators that are defined on them. Figure 1 illustrates the sum operation. Here, and in the next two figures, circles and ellipses depict the operations, and fat arrows show their inputs and outputs. The lilac rectangles are

1 The presence of a colon (:) in a message indicates that an argument must be provided when the message is sent. Thus, colons in Smalltalk play the same role as parentheses and commas in C. The Smalltalk message today printOn: outputStream format: fmt might be represented in Java or C++ as today.printOn_format(outputStream, fmt).

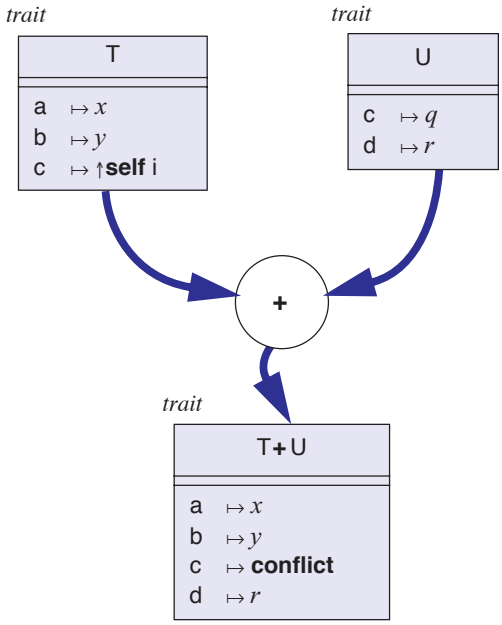


Figure 1. The sum operation on traits.

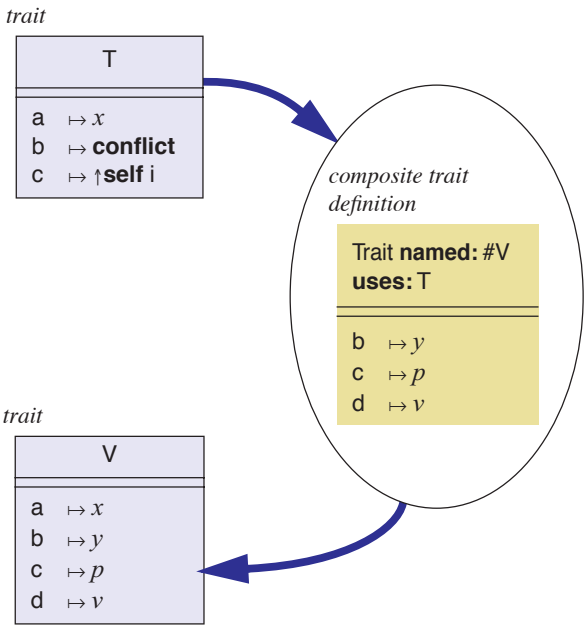


Figure 2. The override operation on traits.

traits[2]; the notation a $\mapsto m$ represents a method with name a and body m. The sum trait T + U contains all of the non-conflicting methods of T and U. However, if there is a *conflict*, that is, if T and U both define a method with the same name, then in T + U that name is bound to a distinguished **conflict** method. The + operation is associative and commutative.

The override operation is shown in figure 2: Trait **named:** #V **uses:** T ... constructs a new composite trait V by combining some explicit definitions (in the pale yellow rectangle) with the existing trait T. The explicit definitions of methods b and c override those from T; the definition of d is added. Because the overriding definitions are given explicitly, it is always clear what is being replaced.

Traits are incorporated into classes by means of an extended form of inheritance. In figure 3, the blue rectangles C and D represent classes. A new subclass D is constructed from a superclass C and a trait T in addition to some explicit local definitions. In Smalltalk we write C **subclass:** #D **uses:** T ...; we call D a *composite class*. Explicit definitions (e.g., of b) override those obtained from T; definitions in T (*e.g.*, of a) override those obtained from C. If necessary, the bodies of the explicit methods (*e.g.*, w) and of the trait methods (*e.g.*, x) can use **super** to call methods in C. In practice, the trait that is used to build a composite class is often the sum of several more primitive traits.

The aliasing operator @ can be applied to a trait to create a new trait that has an additional name for an existing method. For example, if U is a trait that defines methods for c and d, then U@{e → c} is a trait that defines methods for c, d *and* e. The additional method e has the same body as the old method c. Aliases are used to make conflicting methods available under another name, perhaps to meet the requirements of some other trait, or to avoid overriding. Note that because the body of the aliased method is not changed in any way, an alias to a recursive method is not recursive. Finally, a trait can be constructed by *excluding* methods from an existing trait using the exclusion operator −. Thus, U − {c} has a single method d. Exclusion is used to avoid conflicts, or if one needs to reuse a trait that is "too big" for one's application.

Associated with each trait is a set of *required* messages, on which it depends. Any concrete class that uses a trait must provide methods for all of the required messages. For example, if the methods in a trait use the expression **self** size but the trait itself does not define a method size, then size will be in the *requires* set of the trait. When this trait is eventually incorporated into a class that is intended to be concrete, size will have to be defined, perhaps as a method that fetches the value of an instance variable, or perhaps as a method that calculates the size.

Because of the way that we define the operations on traits, the semantics of a method is independent of whether it is defined in a trait T, or in a class (or a trait) that uses T as a component. Consequently, provided that all trait conflicts have been resolved, it is always possible to convert a program that uses traits into an equivalent program that

2 If you are reading a greyscale copy of this paper, we suggest that you obtain a coloured version from one of the authors' web sites.

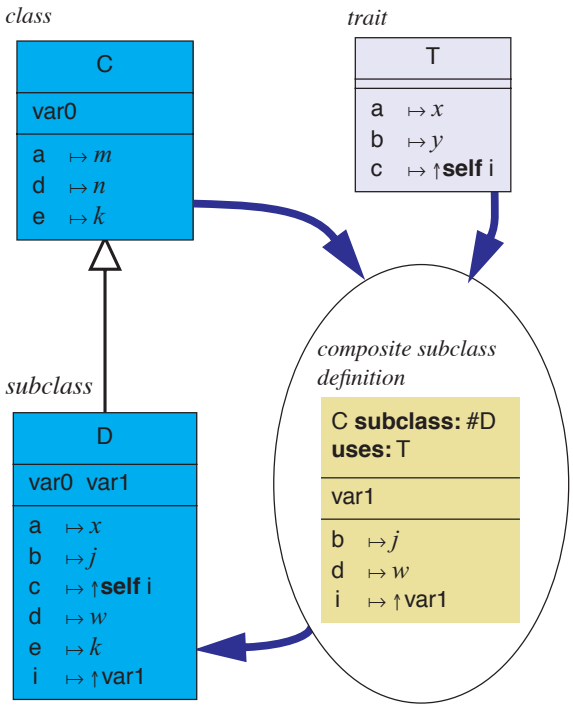


Figure 3. Inheritance using a trait.

uses only ordinary classes, at the cost of possible code duplication. We call this process *flattening*; it is similar to inlining of procedures or expansion of macros. Similarly, a (conflict-free) composite trait can always be flattened into a simple trait. Flattening is illustrated in figure 4: the composite class coloredCircle on the left, composed from two traits and a local definition for draw, is semantically equivalent to the flat class coloredCircle on the right, where the bodies of the corresponding methods are identical.

It is important that flattening never requires the bodies of the methods to be modified. This permits a complex composite entity to be viewed and edited in multiple ways without re-writing its methods. Given appropriate tools, the programmer can choose the structured view, a flattened view in which all the internal structure is elided, or any partially

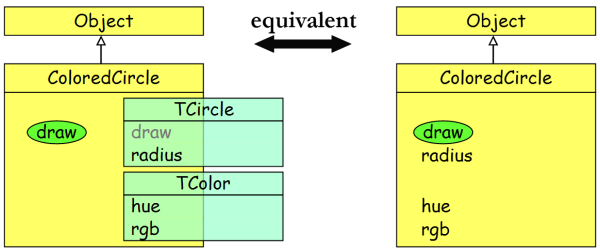


Figure 4. Flattening.

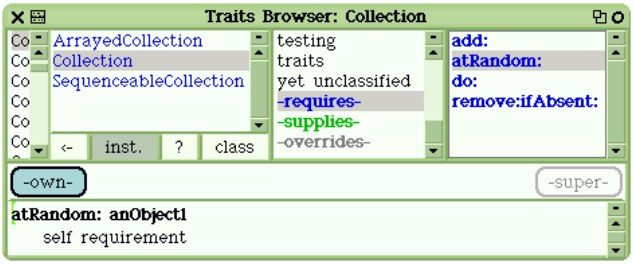


Figure 5. The traits browser.

structured point in between these extremes. We believe that the flattening property is crucial in making traits easy to use; it is another critical difference between traits and mixins.

4. Programming Tools

Although the initial implementation of the trait concept in Squeak Smalltalk took only a few days of pair programming, it quickly became clear that extensive use of traits would require tool support. For example, one of the motivations for traits is the idea that multiple views of a program are better than one, but this idea can only be realized by a programming tool that lets the programmer switch from one view to another without loosing context. Schärli therefore built a tool, the traits browser, to make this possible.

An important feature of the browser is that it automatically and interactively categorizes methods into *virtual categories*, which help the programmer to understand how classes and traits collaborate with each other. As we have described earlier [9], this is valuable information even when no traits are involved and the only relationship between classes is inheritance.

However, as the number of components used to build a class increases, it becomes increasingly important to know how these components collaborate with each other. It is therefore not surprising that the presence of traits not only gives these virtual categories a more sophisticated meaning, but also calls for additional categories. These additional categories result either from dividing existing categories into more fine-grained subcategories, or from unique features of trait composition (*e.g.*, conflicting methods).

4.1. Virtual Categories

Figure 5 shows the traits browser. At first glance it looks like the standard Smalltalk browser[3], but some extra fea-

3 The leftmost pane of the ordinary Smalltalk browser and of the traits browser both contain a classification of classes. Because this classification has no relevance to this paper, to save space when making the figures we reduced the width of this pane so that its contents are unreadable.

tures help the programmer understand the relationships between components. Let us start by summarizing the features that the browser supplies when it is used to examine standard Smalltalk classes.

We have selected class Collection in the second pane in the top half of the browser window. The third pane, which in the standard browser contains a manual categorization of a class's methods, now contains in addition some automatically maintained virtual categories.

The category *-requires-*, which is coloured blue includes all of the messages that the class Collection sends to itself but for which it does not define a method. If there were no such messages, this category would not appear, but Collection is abstract, and requires several methods, which are listed in the fourth pane. We have selected atRandom:, which is consequently displayed in the large pane at the bottom of the browser. The implementation shown, self requirement, is a marker method generated by the browser to indicate that atRandom: is an unsatisfied requirement.

The next category, *-supplies-*, lists methods that are *required* by some other class (or trait) and *provided* by Collection. There is one method in this category, adaptToNumber:andSend:, which shows up here because a FloatArray, a sub-subclass of Collection, super-sends this message, and the intervening class, ArrayCollection, does not define it.

The third category, *-overrides-*, lists those methods provided by Collection that override methods inherited from its superclasses.

A fourth category is not shown in figure 5 because it is empty. This is the category *-sending super-*, which contains all of the methods that make super-sends.

Each of these generated categories has a characteristic emphasis: blue for *requires*, green for *supplies*, grey for *overrides*, and underlined for *sending super*. Even when browsing methods using the ordinary, manually-defined message categories, the names keep their characteristic emphasis. So a supplied method that sends to super will always be shown in green and underlined. The blue colour-coding is also applied to the name of the class itself in the second pane whenever the set of required methods is not empty. This serves as a reminder that the class is incomplete, *e.g.*, it may be an abstract class, or the programmer may still be working on it.

As one uses the browser, even if one is not using traits, one becomes accustomed to the subtle hints provided by these colours and to the instant availability of the virtual categories. They provide valuable reminders of work that remains to be done and of dependencies between classes that would otherwise be invisible.

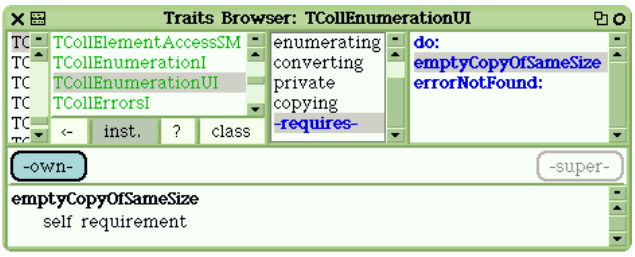


Figure 6. The trait TCollEnumerationUI

4.2. Using the Browser With Traits

The browser displays traits in much the same way as classes, and the virtual method categories described in the previous section are also available for traits. In figure 6 we see how the browser shows the trait TCollEnumerationUI, which encapsulates part of the enumeration protocol discussed in section 2. Note that in order to distinguish traits from classes, the browser displays the trait name in green, whereas a class name is either black (if the class is concrete) or blue (if it is abstract). At a glance, the browser shows us that the trait TCollEnumerationUI *requires* only the methods do:, emptyCopyOfSameSize and errorNotFound: in order to implement all of the methods that it provides. This means that TCollEnumerationUI can be added to any class that provides these three methods. It does not matter whether or not the candidate class is a subclass of Collection.

Composite traits are a little more interesting. Figure 7 shows the composite trait TCollEnumerationI. When this trait is selected, the browser lists its sub-components in the class pane (second from left). This shows the programmer the subtraits from which TCollEnumerationI is composed, and also makes it possible to view the trait in different ways. Selecting TCollEnumerationI shows the flattened view, which contains all the available methods. Selecting -own- shows only the methods defined explicitly in TCollEnumerationI, while selecting TCollEnumerationUI shows the methods reused from this subtrait.

Since multiple subtraits are composed with the commutative sum operation, not with inheritance, the categories *-supplies-* and *-overrides-* have a slightly different meaning for traits than for classes. The *-supplies-* category contains the methods that are required by a subtrait and are provided by the currently selected subcomponent. In figure 7 we have selected -own-, and therefore the *-supplies-* category contains the methods required by the subtrait TCollEnumerationUI and implemented by TCollEnumerationI itself. Similarly, the category *-overrides-* shows the methods that are provided by a subtrait and then overridden by an "own" method of the composite trait.

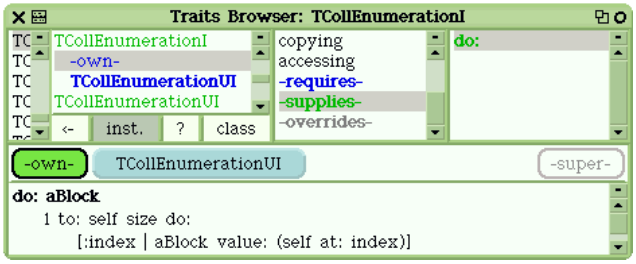


Figure 7. Composite trait TCollEnumerationI**.**

In a class built using traits, the *-overrides-* virtual category may contain two kinds of methods: those that override methods of the superclass and those that override methods of a subtrait. To make this distinction visible to the programmer, the *-overrides-* category of a composite class consists of two subcategories: *super* and *traits*. This is also the case for the other virtual categories.

In addition to the four virtual categories described in section 4.1, there is also a fifth virtual category: *-conflicts-*. This category applies only to entities composed from traits, and lists the methods defined by more than one component trait: these are conflicts that the programmer must resolve.

The "button bar" in the centre of the browser serves the dual function of showing the programmer which version of a method is on display, and allowing the selection of another version. The black border on the -own- button in figure 7 indicates that TCollEnumerationUI's *own* version of do: is on display; clicking the button TCollEnumerationUI would switch the display to the version obtained from that trait.

According to our usual colour scheme, the blue colour of the button TCollEnumerationUI and the green colour of the button -own- show that the method do: is required by the subtrait TCollEnumerationUI and supplied by the composite trait itself. The buttons are arranged from left to right according to the precedence that follows from the trait composition rules: "own" methods override methods from subtraits, which in turn override methods inherited from the superclass. Thus, the leftmost button that is active (*i.e.*, that is not greyed out) corresponds to the currently applicable method.

5. Programming Methodology

Most class-based languages overload the class concept with too many responsibilities. In his thesis [3], Bracha lists no less than 11 distinct roles for classes. At a coarser granularity, we distinguish 5 roles for classes that are relevant to the use of traits:

1. conceptual classification of objects,
2. definition of protocols (interfaces) for objects,
3. modularization — the grouping of related methods,
4. reuse (sharing) of implementation, and
5. incremental modification of an existing class.

It is often difficult, and sometimes impossible, to make a single class hierarchy play all of these roles. Usually, it is the conceptual relationship between the class hierarchy and the domain that suffers, because corrupting this relationship does not immediately break the program. In the case of the Smalltalk collection classes, Cook has shown how the inheritance hierarchy fails to capture the conceptual relationships between the various collections: the conceptual hierarchy has been subverted to allow greater reuse [4]. For example, whereas Dictionaries are conceptually a kind of Updatable Collection, they are implemented as a subclass of Set. The problem with subverting the inheritance hierarchy in this way is that the code no longer models the domain, and thus it is likely to be more difficult to understand, and harder to modify in response to changes in the requirements.

A trait-based programming methodology avoids this problem. Traits support modularization directly (role 3), and methods encapsulated in a trait can be reused at any point in an inheritance hierarchy (4). The inheritance operation on traits provides for incremental modification of an existing class and for the reuse of the "delta" (5). Traits concretize the otherwise abstract notion of protocol, and thus make it much simpler for a number of classes to define the same interface, whether or not they are related by inheritance (2). This frees the class hierarchy to be used for conceptual classification (1).

In the remainder of this section, we explain in more detail how traits change the programming process and how the traits browser supports the new process.

5.1. Uniform Protocols

In section 2, we explained why it is so important for disparate classes to understand a uniform protocol. In a conventional class-based language, the only tool available to induce this uniformity is inheritance. If inheritance is used for another purpose, the programmer must instead construct the protocols "by hand," one method at a time. In addition to the dangers of code duplication, protocols constructed by hand are unlikely to stay uniform: over time, one of the classes is likely to be extended while the other is forgotten.

Traits solve this problem by making it possible to construct classes by trait (*i.e.*, protocol) composition as well as by inheritance. Given perfect foresight, any protocol that must eventually be supported by disparate classes can be implemented in a trait, and re-used wherever it is needed. Unfortunately, mere mortals tend to have difficulty applying methodologies that rely on perfect foresight. Instead, we allow programmers to build classes in the conventional way, implementing protocols by placing methods directly in

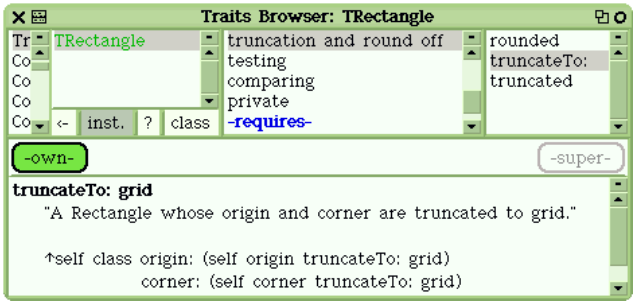


Figure 8. The trait TRectangle

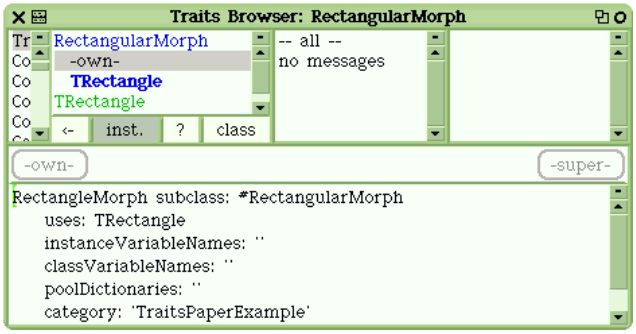
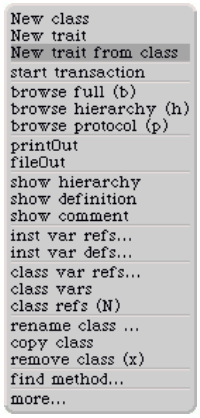


Figure 9. The new class RectangularMorph **immediately after its creation.**

whichever class needs them. If it becomes apparent that the same protocol needs to be supported in an additional (and unrelated) class, we provide a tool that enables the programmer to extract the protocol into a trait, and then to use this trait as a component of both the original and the additional class.

For the sake of concreteness, we will consider the Squeak class RectangleMorph. RectangleMorph is a subclass of Morph, which is the root of the GUI hierarchy in Squeak. RectangleMorph does not understand the protocol of class Rectangle. This is a problem because RectangleMorph *looks* like a rectangle, and it also contains within it the state required to *be* a rectangle. A user will reasonably expect it to understand the protocol of a Rectangle. This requires that 70 additional methods be added to RectangleMorph.

We can easily create a trait that contains these 70 methods by extracting them from class Rectangle. The "yellow button" contextual menu available in the class list pane of the browser (shown on the right) gives access to a number of useful commands. The menu item "*New trait from class*" generates a template that we use to build a new trait from class Rectangle and to name it TRectangle. When we accept the template, the browser creates copies of all of Rectangle's methods, abstracts the references to Rectangle's two instance variables (origin and corner), and populates the trait TRectangle with the new methods. The result is shown in figure 8.



Notice the effect of the *abstract variable* refactoring on the displayed method: whereas the truncateTo: method in class Rectangle accessed the instance variables origin and corner directly, the version in TRectangle sends the messages self origin and self corner.

The *-requires-* virtual category of TRectangle contains three methods: origin, corner and species. These methods represent the places where the trait TRectangle must connect to any class in which it is used; they are in effect parameters of the trait.

Continuing with the example, we again use the yellow button menu, this time to pull up the template for defining a new class. This is like the class creation template in ordinary Smalltalk, but requests an additional parameter: the traits to be used as components of the new class.

The state of the browser once this template has been completed is show in figure 9. The class RectangularMorph has been created, but its name appears in blue, showing that it is incomplete, *i.e.*, that some of its requirements are unsatisfied. When the name of a class is selected, a list of its components appears indented beneath it in the class pane. The *-own-* pseudo-component contains those methods that are defined directly in the selected class; in figure 9, *-own-* is empty, since we have not yet written any methods for RectangularMorph. The component TRectangle contains all of the methods that we placed in the trait TRectagle in the previous step; it too is blue, showing that it also has unsatisfied requirements. For each non-empty component, all of the non-empty virtual method categories *-requires-*, *-overrides-*, *-sends-super-*, *etc.* are shown in the method category pane.

We can use these virtual categories to find the unsatisfied requirements. Moving down to the *-requires-* category of TRectangle, we can view and edit the self requirement marker methods. For example, we can define

```
corner
    ↑ self bounds corner
```

and similarly for origin. These new methods populate the *-own-* component of the class RectangularMorph. This is because we defined the methods while browsing RectangularMorph; if instead we had been browsing TRectangle, then the methods would have populated that

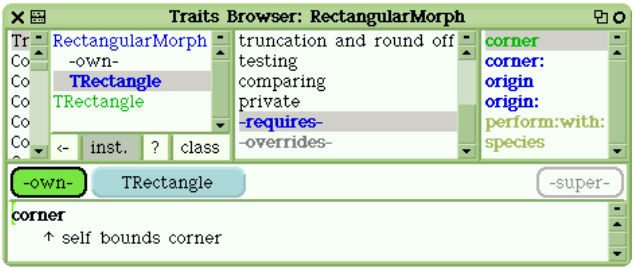


Figure 10. Method corner **in** RectangularMorph

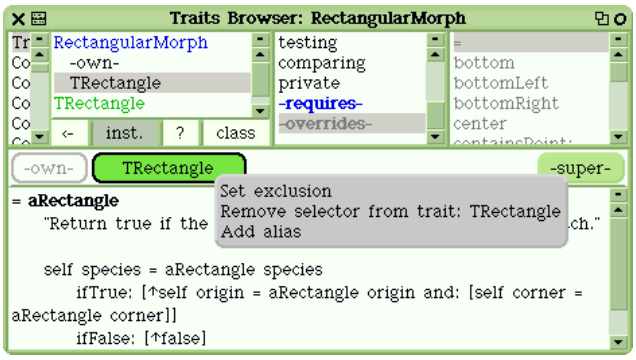


Figure 11. Method = **in** RectangularMorph**.**

trait. Once a required method has been defined, the corresponding selector remains in the *-requires-* category of the trait, but turns green because it is now *supplied* by the class in which the trait is used (see figure 10).

The species requirement of TRectangle was green from the first because RectangularMorph inherits a species method from Object. It is important that satisfied requirements remain visible: the list of required methods is a useful aid to understanding the dependencies inside a class, whether or not they have been satisfied. However, once all of the requirements have been satisfied, the name TRectangle changes from blue to black.

The last step in creating RectangularMorph is to examine the places where we have overridden methods inherited from RectangleMorph with methods from trait TRectangle. These methods are listed in the *-overrides-* virtual category. The "button bar" in the browser (see figure 11) lets the programmer view both the superclass and the trait methods for the currently selected message. If several traits had been used as components, there would be a button for each; an *-own-* button is also available if the class defines a method locally. Using these buttons, the programmer can easily view the various competing methods, and decide which is appropriate for the new class.

In the RectangularMorph example, most of the overrides provided by the trait are appropriate, but =, hash and printOn: are not. A browser menu (see figure 11) gives us a choice of two ways to exclude these methods. "*Set exclusion*" modifies the definition of RectangularMorph so that the selected method (=) is excluded from the composition. If we were to use this menu item three times, for methods =, hash and printOn:, the browser would modify the definition of RectangluarMorph to read as follows.

```
RectangleMorph subclass: #RectangularMorph
    uses: TRectangle – {#=. #hash. #printOn:}
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'TraitsPaperExample'
```

In these three cases, the more appropriate action is to *remove selector from trait TRectangle*. That is, we see that these methods should not be in the trait TRectangle at all: by removing them from the trait, they no longer override the inherited methods in RectangularMorph.

Note that the browser lets us view and edit our new class in two ways. By selecting the name RectangularMorph in the browser, we can view it as a conventional Smalltalk class. Alternatively, by selecting its components -own- and TRectangle, we can view it as a composite entity. Editing a method, in either view, changes just the class RectangularMorph: even if the method originally came from the trait TRectangle, a modified version is created for the class. Deleting a trait method sets an exclusion, so that the method is no longer part of the composite class. If instead the programmer wants to modify the component trait, by editing or deleting one of its methods, the browser must first be focussed on the trait.

Now RectangularMorph is complete, but our task is not yet finished, because we have duplicated all of the methods that we extracted from Rectangle when we constructed TRectangle. We must eliminate this duplication by refactoring the Rectangle class so that it uses TRectangle.

5.2. Uncovering Hidden Structure

One of the more difficult tasks of program maintenance is discovering the latent structure hidden in the old code. The complete protocol of a class is usually the union of several smaller protocols, but programmers seldom make these component protocols explicit. The standard Smalltalk browser allows the programmer to categorize methods into protocols, but this is mere documentation, and thus often wrong: getting the categorization right has no immediate payoff. Java provides **interfaces** and the **implements** declaration, but their use is optional: it is more common to program to the implicit interface of a whole class. The traits browser has proved to be a powerful tool for modularizing a class into separate traits, each of which implements a coherent protocol.

The first major programming task that we undertook with the browser was the refactoring of the Smalltalk collection classes into traits [2]. This required us to discover the various protocols that were understood by the existing collection classes. We developed a methodology that involved two copies of the browser.

In the first browser we took a class from the existing hierarchy, say, Collection, and extracted a trait from it, which we called TempCollection, and which thus contained copies of all of the methods of Collection. We then moved (by dragging) the methods from TempCollection into an appropriate trait in the second browser. If an appropriate trait did not yet exist, we created one. For example, we might drag methods like collect: into a new trait called TCollEnumeration. The methods do: and species would then immediately appear in the *-requires-* virtual category, coloured blue.

The immediate updating of the *-requires-* category provided valuable feedback about the protocol that we were constructing. For example, if the method addAll: were mistakenly dragged into TCollEnumeration, then add: would immediately show up in the *-requires-* category, providing a strong clue that something was amiss. If necessary, the menu item "*local senders of . . .*" let us see why a particular method was required.

Sometimes we would realize that what we had imagined as one trait was actually two, in which case we would pause in our work while we split the trait. For example, we eventually split TCollEnumeration into two traits: TCollEnumerationUI, whose methods will work on unsequenced collections, and TCollEnumerationI, which contains TCollEnumerationUI as a subtrait, but which also includes some methods (like findFirst: and from:to:do:) that require the collection to be sequenced, and which thus have at: as a additional requirement. Making *-requires-* visible enables the programmer to see this distinction.

The process of discovering the latent protocols embedded in Collection proceeded until there were no more methods in TempCollection. At this point we could delete the temporary trait, confident that all of its methods had found a home in some trait or other.

If the new traits that are constructed by this process are to be reusable, the semantics of each of the required methods must be clear. The programmer can document these semantics by converting the automatically generated **self** requirement marker method into a **self** explicitRequirement method, and adding a comment describing the required behaviour. The few cases in which we were unable to understand the required semantics represented design flaws in the original code. For example, we have already stated that several of the methods in TCollEnumerationUI required species. But so did = and hash! What is going on here is that species is actually playing two different roles. According to the Blue Book[6], **self** species new should return an instance of a collection "similar to" **self**. But according to LaLonde and Pugh[8], species also plays a critical role in equality comparisons: two collections must be of the same species if they are to be considered =. These two roles are not always compatible, so part of our refactoring was to replace occurrences of **self** species new (the first role) with a self-send of the new message emptyCopyOfSameSize.

5.3. Traits and Agile Methodologies

In recent years, agile methodologies, in particular Extreme Programming (XP) [1], have have begun to influence the software engineering process. Traits and the traits browser are compatible with several of the XP practices: continuous design, constant refactoring, testing, pair programming, and collective ownership.

Continuous Design. Extreme programming suggests that there is no up-front design phase. Instead, design takes place incrementally throughout the development process: the design of a program is always subject to change. Traits support this style of programming because they provide an additional way to adapt a program to a design change. Specifically, in addition to refactoring the class hierarchy in the conventional way, traits allow one to factor out an arbitrary set of methods and then reuse them wherever it seems most appropriate.

Traits even allow one to start implementing before any design exists. This is because traits enable a *behaviour-based* or bottom up strategy that is appropriate when it is clear that a certain behaviour is needed, but not yet clear in what class it should be placed. The enumeration behaviour already discussed is a typical example. Traits let us forge ahead and define as a trait the coherent set of methods that captures the appropriate behaviour; we can defer the decision about where it should be placed in an inheritance hierarchy to best enable reuse.

Of course, it is still possible to adopt a *class-based* or top-down strategy: when it is clear that some behaviour has to be in a class, we can still use all the familiar techniques of single-inheritance programming. We can just implement a class, or a small hierarchy of classes, as if traits did not exist. Later, we can structure the classes by dragging and dropping certain methods into traits. Perhaps this is just for documentation, but it can also be essential if we eventually see that one of the classes contains some reusable behaviour that we would like to share with an unrelated class.

Refactoring is the technology that makes continuous design feasible: "if you believe that the future is uncertain, *and that you can cheaply change your mind*, then putting in functionality on speculation is crazy" [1, page 57, our emphasis]. Refactoring is simplified by the presence of traits, because they enable us to move a whole group of logically

related methods from one class to another with a single edit. By putting methods into traits, we keep our options open: if there later turns out to be an abstract superclass that is a suitable home for such behaviour, then that superclass can use the trait, and its subclasses can inherit the corresponding behaviour, without making it any harder for other, unrelated classes to also use the same behaviour. This flexibility seems to have no cost in understandability: indeed, we argue that traits *increase* understandability. This is because critical behaviour can be made explicit and given a descriptive name.

Testing. Another advantage of traits is that they allow one to specify tests in a more fine-grained and reusable manner. This is because tests can be associated with traits as well as with classes, and traits represent a smaller and more primitive unit of functionality. Even for very simple traits such as equality, magnitude, or emptiness, tests can be written very early, placed in the trait, and then applied to all of the classes that use the trait.

Pair Programming— two programmers working together at one keyboard — also proved to be a valuable technique during our application of traits to the Smalltalk collection classes. Having such a fine-grained way of implementing and reusing behaviours made our development cycles short enough that the implementation and design steps began to overlap. It was very valuable for us to be able to discuss the innumerable small design decisions with each other as soon as they arose. Pair programming also helped to keep us honest, encouraging us to use traits to perform "mini-refactorings" as soon as either member of the pair noticed that an improvement of the code was possible.

Collective ownership of code becomes even more important when programming with traits. In traditional class-based programming, it is possible to assign responsibility for individual classes, or for small hierarchies, to different programmers. Assigning ownership in this way would severely limit the utility of traits, because when a trait is extracted from some other class, it is necessary to refactor that class so that it uses the new trait. With collective ownership, refactoring any class in the hierarchy is not only permitted but encouraged, so long as the *total* amount of code is reduced.

5.4. Interaction between Tools and Methodology

We believe that tools and methodologies must be developed together: the best methodology is little more than pious hope without supporting tools, and powerful tools can be powerfully dangerous without a guiding methodology. In this subsection we explore some of the ways that our tools encourage the development of sound methodology. Note that by the term "tools" we include traits as a programming language feature as well as the traits browser: the programming language is a programmer's most important tool.

The difference between traits on the one hand and mixins and multiple inheritance on the other arise from the way in which we defined the composition operators on traits. These operations ensure that conflict resolution is always explicit, and that a composite entity can always be "flattened" into a simple, unstructured one. We will now explain why we feel that these properties are so important.

We have previously argued that the complex rules for conflict resolution that accompany many schemes for multiple inheritance, and the implicit and "silent" resolution of conflicts that is characteristic of mixins, are a source of unexpected behaviour and a major reason that these technologies are usually avoided [10]. In contrast, when a new class (or a new trait) is built from two or more component traits using the associative and commutative + operation, any method with different definitions in the components results in a trait conflict. The browser generates a marker method with body `self traitConflict`, places the selector in the virtual category *-conflicts-*, and colours it red.

It is the programmer's responsibility to resolve each of these conflicts explicitly, *i.e.*, to empty the *-conflicts-* category. This can be done by modifying the component traits, by excluding a particular method from the composition (*e.g.*, using the "*set exclusion*" menu), or by replacing the marker method with a completely new method that overrides the conflicting alternatives. The row of buttons above the code pane helps in this process, because it contains a button for each of the available implementations of a method. The colour and emphasis of these buttons indicate the status of the corresponding methods, for example, required methods have blue buttons and excluded methods have semi-transparent buttons.

Once all of the conflicts have been resolved, the meaning of the program should be clear even to a casual observer, who does not have to be familiar with complex disambiguation rules.

An interesting situation is when two component traits A and B both use trait T as a sub-component, and thus both define all of T's methods. In this case, there are no conflicts due to T's methods in A+B, because the definitions in A and B are identical. It has been argued that this is a mistake, because it exposes the internal structure of A and B, but we do not feel that this is a problem: traits, like classes, are intended as white boxes, not black boxes. However, if A is changed (to A') so that A' overrides some of T's methods, then all of the overrides will show up as conflicts in $A'+B$, and must be resolved by the programmer. We feel that this is timely: the programmer is asked to resolve a conflict as soon as it appears, when the information necessary to do so is most likely to be in short-term memory. Treating A + B as containing conflicts would require that the conflicts be re-

solved before they were real, and thus before the programmer had enough information to resolve them correctly. It seems more likely that the choice would be made arbitrarily; consequently, when A is changed to A′ the real conflicts would not be flagged.

The reason that the flattening property is so important is that we want traits to support two different views of a program — a flat or class-based view and a structured or trait-based view. This means that the programmer can choose to view a program in a way that hides all of the traits and presents only conventional Squeak classes. Once again, we emphasize that this works because of two details of trait composition:

1. **super** in a trait method is bound only when the trait is used to form a subclass, and *not* when it is composed with other traits, and
2. there is no "deep rename" operation on methods; the *alias* operation gives a method an additional name, but does not change its body.

Flattening also makes it reasonable to build a class from *scores* of traits. The user of such a class can ignore the internal structure: it is no more complex than a class with the same methods built "from scratch" in conventional Smalltalk, and can indeed be viewed in an identical way in our browser. However, the extra internal structure is available, if and when it is wanted, to assist in understanding and reuse. In contrast, classes built from mixins cannot be flattened without changing the semantics of the methods. In practice, this limits the number of mixins per class to a handful.

6. Conclusion

As we had hoped, we found traits and the traits browser to be valuable tools because they provide multiple alternative views on a program.

However, in addition to this expected benefit on program understanding, we also found that traits enable us to build classes in new ways. In essence, traits let us delay making a decision until we have enough information to make it correctly: traits promote late binding much better than conventional object-oriented development. What makes it easy to switch between the trait-full and trait-less styles is the flattening property: the fact that the semantics of a method is independent of whether it is defined in a trait T, or in a composite entity (a class or another trait) that uses T.

Once a selection of relevant traits had been built up, we also found that traits raised the level of abstraction at which we programmed quite significantly. Writing classes became as simple as identifying the right traits, building a class that used them, and filling in the instance variables and the methods that connected the traits to those variables.

This is exactly how we proceeded towards the end of our refactoring of the collection classes. We combined the traits; the browser showed us what was missing; we filled in the missing things and found that the class was working *immediately*!

The properties of trait composition are important in making this process work. Because the sum operation is commutative, because conflicts have to be resolved explicitly, and because there is no state, this style of programming does not usually create buggy code, or code that has too much state. The browser shows the conflicts, the requirements and how the traits are connected to each other. In addition, it shows all the overrides: provided that the programmer checks the overrides and the new methods, the composite class has to do the right thing.

Acknowledgments. We are indebted to Gilad Bracha for insightful discussions about conflict resolution, and to our colleagues at OGI and IAM, particularly Stéphane Ducasse and Roel Wuyts, for their critical evaluation of the traits browser.

References

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.

[2] A. P. Black, N. Schärli, and S. Ducasse. Applying traits to the Smalltalk collection hierarchy. In *Proceedings OOPSLA '03*, pages 47–64, Oct. 2003.

[3] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Ph.D. thesis, Dept. of Computer Science, University of Utah, Mar. 1992.

[4] W. R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, volume 27, pages 1–15, Oct. 1992.

[5] E. W. Dijkstra. Notes on structured programming. In E. Dijkstra, O.-J. Dahl, and C. A. R. Hoare, editors, *Structured Programming*, pages 1–82. Academic Press, Inc., New York, NY, 1972.

[6] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.

[7] M. Guzdial. *Squeak — Object Oriented Design with Multimedia Applications*. Prentice-Hall, 2001.

[8] W. LaLonde and J. Pugh. *Inside Smalltalk: Volume 1*. Prentice Hall, 1990.

[9] N. Schärli and A. P. Black. A browser for incremental programming. *Computer Languages, Systems and Structures*, 2004. (In press, special issue on Smalltalk).

[10] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.