

# A Browser for Incremental Programming

Nathanael Schärli and Andrew P. Black

*Software Composition Group, University of Bern, Switzerland*

*OGI School of Science & Engineering, Oregon Health and Science University, USA*

---

## Abstract

Much of the elegance and power of Smalltalk comes from its programming environment and tools, but despite the passage of more than 20 years, the Smalltalk browser is much the same today as when it was first introduced. We have extended this browser with features that dramatically improve its support for incremental programming by employing real-time analysis of the code being modified. We illustrate these improvements by example, and summarize the algorithms used to implement them efficiently.

*Key words:* Smalltalk browser, incremental programming, intentional programming, method reachability, requires set, trait conflict.

---

## 1 Introduction

The most important of the Smalltalk programming tools is the *Browser*, which allows the programmer to examine, modify and extend Smalltalk’s object-oriented code base. The Browser was revolutionary when it was introduced, but over the intervening 20 years it has not been much improved. Useful extensions for semi-automated refactoring have been added, leading to a tool known as the Refactoring Browser [9]. Apart from that, today’s Smalltalk Browser is essentially the same as it was in 1980.

In the meantime, the concept of the “Integrated Development Environment” (IDE)—for that is what the Smalltalk toolset would now be called—has proved to be so successful that similar environments have been created for other programming languages. For example, IBM’s VisualAge for Java [6] was essentially a re-targeting of the Smalltalk IDE to Java; more recently the cross-language environment Eclipse [3] has been making similar tools available for many popular languages.

The lack of development of the Smalltalk Browser is even more surprising when we recall that Smalltalk was designed to support incremental and experimental programming [14], and that it encourages a style of programming in which classes are

---

*Email address:* `schaerli@iam.unibe.ch`, `black@cse.ogi.edu` (Nathanael Schärli and Andrew P. Black).

created frequently and methods are kept small [7]. Thus, the usability of Smalltalk depends on the development tools more than does the usability of other languages.

Today’s Smalltalks run on a commodity laptop about a thousand times faster than they ran on commodity machines of the early 1980s, and about fifty times faster than on the custom hardware of the Dorado. This power has been harnessed to write Smalltalk applications for real-time music and motion picture manipulation that would have been unthinkable twenty years ago. But the power has *not* been used to build tools that significantly improve the programming process. We believe that this is a lost opportunity.

In this paper, we present one such tool. It derives its power from real-time analysis of the code being modified, which enables us to infer system-wide information about the structure of the program. We have found that this allows a programmer to gain a better understanding of the program, and actively supports incremental programming.

The basic idea is to compute and display a real-time classification of the methods of a class  $C$  into “virtual categories”. One of the most useful is the *requires* category, which lists all of the methods that are sent to **self** by other methods of  $C$ , but which are neither explicitly defined nor inherited. We also display the *supplies* category, which contains the concrete methods of  $C$  that implement inherited abstract methods, and the *overrides* category, which contains the concrete methods of  $C$  that override inherited concrete methods. The last two virtual categories that we support are *calls super*, which lists the methods that perform super-sends, and *conflicts*, which we will explain in section 3.3.

Of these virtual categories, the first—the real-time computation of the set of required methods—is the one that we have found most useful. The display of this category supports “programming by intention” [5], a top-down style that encourages the programmer to think about *what* has to be done rather than about *how* to do it. The idea behind programming by intention is to imagine methods that do “the hard part” of one’s task, so that all that one has to do to complete the task is send those messages. Of course, later one applies the same idea to defining the “hard” methods. In this setting, the *requires* category provides a constant reminder of what is left for the programmer to do.

We have implemented all of these virtual categories in a browser for Squeak, an open-source dialect of Smalltalk [4,13]. In addition to being the most useful, the *requires* category also turns out to be the trickiest to define and implement; to the best of our knowledge there is no other browser that displays it. In contrast, the definition and computation of the other virtual categories is mostly straightforward, and recent versions of other Smalltalks (*e.g.*, VisualWorks [15] and Squeak [13]) also compute some similar properties and use the results to decorate method names in the browser. However, no other browsers use this information to group methods

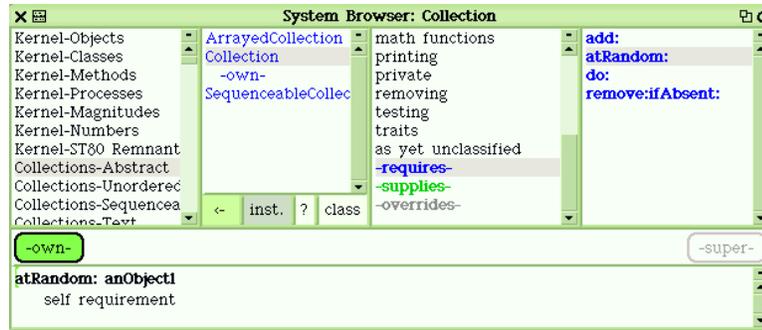


Fig. 1. The traits browser examining the class Collection.

into virtual categories that supplement the programmer-defined categories and allow the programmer to understand the class from different viewpoints. A Squeak image that demonstrates our browser, and contains its source code, is available [10].

This paper makes two contributions. First we report on our practical experience using the extended browser and how it changes the programming process in two environments: ordinary Smalltalk (section 2) and Smalltalk with traits [11] (section 3). Second, we describe how we implemented the computation of the *required* set efficiently enough for it to be used in real-time (section 4).

## 2 Using the Browser

Our browser is shown in figure 1. At first glance it looks like the standard Smalltalk browser, but a few extra features are visible. In the figure, Collection is selected in the class pane (the second from the left). The third pane, which in the standard browser contains a manual categorization of the methods of the selected class, now contains in addition some virtual categories. The category *-requires-*, which is colored blue, includes all of the methods that the class Collection sends to itself but does not define or inherit. If there were no such methods, this category would not appear, but Collection is abstract, and requires several methods to be complete. The list of these required methods appears in the fourth pane. In the figure 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 other three requirements, `add:`, `do:` and `remove:ifAbsent:` do in fact have implementations: the marker method `self subclassResponsibility`. However, the browser recognizes this as not being a real method and still categorizes them as required.

The next category, *-supplies-*, lists methods that are required by some other class (or trait, see section 3.1) and provided by the class that we are browsing. There is one method in this category, `adaptToNumber:andSend:`, which shows up here because a method in class `FloatArray`, a sub-subclass of `Collection`, `super-sends` this message. `Collection` defines it, while the the intervening class, `ArrayedCollection`, does *not* define it. This can all be ascertained by using the standard *implementors of ...* browser.

The third category, *-overrides-*, lists those methods provided by `Collection` that override methods inherited from its parent. The *-sending super-* category is not shown in the figure because it is empty.

Although it is not clear from the grayscale figure, each of the generated virtual 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 method categories, the names keep their characteristic emphasis. So a supplied method that sends to super will always show up in green and underlined. The blue color-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, one grows accustomed to the hints provided by these colors and the availability of the generated categories. They remind one of work that remains to be done, and of otherwise-invisible dependencies between classes.

It is critical that the virtual categories are recomputed in real-time. Many errors can be detected because classes become “blue” unexpectedly; when one looks to see why, one finds, for example, that the class has acquired a weird requirement that is actually caused by a typographical error. The browser has also shown us that many classes are “accidentally” abstract, such as `Morph` and all its subclasses.

Additionally, the browser helps to prevent “inter-level” errors, such as removing a method from a class where it is not used, without noticing that the method is still used by a subclass. In statically compiled languages, this sort of bug can be caught by compiling the whole program. The error detection provided by the traits browser is not as complete, but it disrupts the programming process far less, and is thus better suited for experimental programming.

With respect to program understanding, we have found that the alternative view of a class obtained from looking at the *requires* category gives us real insight into why a class is abstract. For example, looking at `Collection`, we can apply the *local senders of...* enquiry to the required method `do:` and learn that it has 33 senders in class `Collection`, all of which will be available on any subclass of `Collection` that correctly implements `do:`.

We also noticed that the methods that depend on `do:` are mostly pure behaviour, that is, they send messages but do not directly access any variables. Thus, they would work correctly on any class that supplied `do:`, not just on subclasses of `Collection`. We realized that such a group of methods, encapsulated as unit of reuse, would provide a much more flexible and fine-grained way of sharing code than inheriting a whole class. This realization was the key to a language extension that we have developed called traits [11].

### 3 Using the Browser With Traits

#### 3.1 What Are Traits?

A trait is a named group of *pure* methods, *i.e.*, methods that do not access any instance or class variables. A trait appears in the browser very much like a class, except that a trait has no superclass and no variables. In fact, the browser provides a menu item that extracts a trait from a class by applying the *abstract variable* refactoring [9] to all concrete variable references in the methods of the class.

Those familiar with the concept of a mixin [2] will notice that a trait is like a simple kind of mixin without any dependencies on variables. The main difference is that a simple but powerful set of combinators lets us use traits to build other traits and allows the methods in a trait to be reused in classes outside of the inheritance hierarchy. The comparison with mixins is pursued elsewhere [11]; a formal model for traits is also available [12].

A good example of a trait is `TCollEnumerationUI`, which encapsulates the enumeration protocol for collections. This trait was constructed during our refactoring of the collection classes [1], and provides the following methods:

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

To accomplish this, `TCollEnumerationUI` *requires* the methods `do:`, `emptyCopyOfSameSize` and `errorNotFound:` — and no others. For example, `allSatisfy:` is defined as follows:

#### **allSatisfy: aBlock**

"Evaluate aBlock with the elements of the receiver. If aBlock returns false for any element return false. Otherwise return true."

```
self do: [:each | (aBlock value: each) ifFalse: [↑ false]].  
↑ true
```

This means that the trait `TCollEnumerationUI` can be added to any class that provides the three required methods. It does not matter if the candidate class is a subclass of `Collection`. For example, the class `LayoutCell`, used to lay out components in the Morphic graphical interface, is a direct subclass of `Object`; it implements `do:`, but not the other enumeration methods. (The single exception is `inject:into:`, the method for which has been copied from class `Collection`.)

The trait `TCollEnumerationUI` can be added to `LayoutCell`, so long as the other required methods are provided. Thus, by writing a minimal amount of new code, the

protocols of classes can be made more uniform with those that already exist, even if it is impossible or inappropriate to inherit all of the existing behaviour.

We developed the ideas for this browser shortly after we began working on traits. We found that the more components that are used to build a class, the more essential it becomes to know how these components are “parameterized”, that is, which methods are like do: in the above example. When more than one trait is combined, is also important to know whether any methods conflict, and which methods the class adds locally, either to satisfy the requirements of the traits that it uses, or to implement its own unique behaviour.

Armed with this realization, we developed the traits browser to provide this information in an interactive way. Subsequently, we noticed that many of the features that we added to improve the usability of traits are also really useful when applied to ordinary single-inheritance classes.

### 3.2 Working with Traits

Now let us explore some of the facilities provided by the Browser for working with traits. For the sake of concreteness, let’s improve RectangleMorph, a subclass of Morph. 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 exhibit the behaviour of a Rectangle.

Seventy additional methods are required to make RectangleMorph a Rectangle. To add these methods without duplicating code, we first create a trait that contains these methods, by extracting them from the existing code for Rectangle.

The “yellow button” contextual menu available in the class list pane of the browser gives access to a number of useful commands. The menu item “New trait from class” enters a template in the bottom pane of the browser. We can use this template to build a new trait from class Rectangle and to name it TRectangle. 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 *-requires-* category of TRectangle lists three methods: origin, corner and species. These required 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 very much like the subclass creation template in ordinary Smalltalk, but contains a place for an additional parameter: the traits to be used in constructing the subclass.

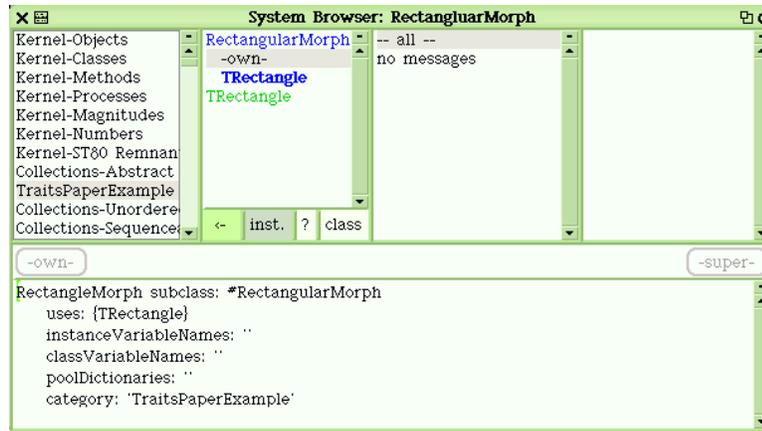


Fig. 2. The new class RectangularMorph immediately after its creation.

The state of the browser once we have accepted this template is shown in figure 2. 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 2, *-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 *TRectangle* in the previous step; it too is blue, showing that it also has unsatisfied requirements. For each non-empty component, all of the virtual method categories *-required-*, *-overrides-*, *-sends-super-*, *etc.* are shown in the method category pane.

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

```
corner
  ↑ self bounds corner
```

and similarly for *origin*. These new methods populate the *-own-* component of the class RectangularMorph. Once a required method has been defined, the corresponding selector in the requires category turns green. It is important that a requirement does not “go away” just because it is satisfied: the list of required methods is a useful aid to understanding the dependencies inside a class, whether or not they have been satisfied. Once all of the requirements have been satisfied, the name *TRectangle* also changes from blue to black.

Note that the Browser lets us view 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 structured entity.

The last step in creating RectangularMorph is to examine the places where we have

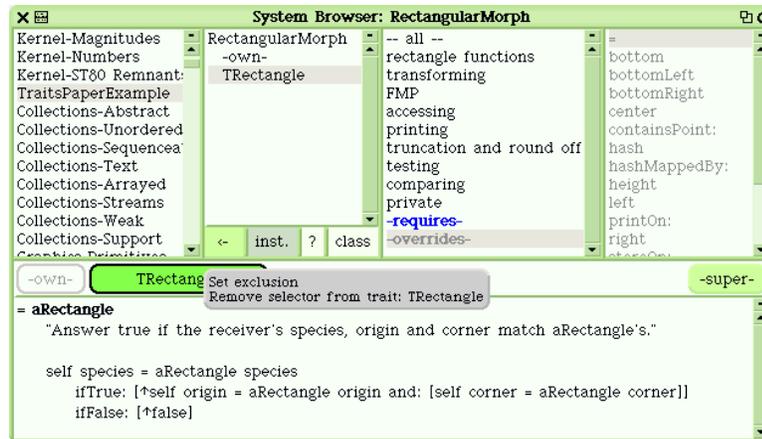


Fig. 3. The Browser is used to examine methods from TRectangle that override methods inherited from RectangleMorph.

overridden methods inherited from RectangleMorph with methods from trait TRectangle. These methods are conveniently listed in the virtual category *-overrides-*. A row of buttons in the browser (see figure 3) lets the programmer view the inherited and the trait methods. If several traits had been used to compose the class, 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 switch from one version of a method to another, and decide which is appropriate for the new class.

In this example, most of the overrides provided by the trait are appropriate, but `=`, `hash` and `printOn:` are not. A browser menu (see figure 3) gives us a choice of two ways to exclude these methods from RectangularMorph. “Set exclusion” modifies the command that is used to build RectangularMorph so that the selected method (in this case `=`) is excluded from the composition. If we use this menu item three times, for methods `=`, `hash` and `printOn:`, the browser will modify the definition of RectangularMorph to read:

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

In this particular case, the more appropriate action is to “remove selector from trait TRectangle”. That is, we decide 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.

Now RectangularMorph is complete, but our task is not quite finished, because we have introduced some code duplication. We need to go back and refactor class Rectangle, so that it also uses trait TRectangle.



Fig. 4. Compiling an accessor for an instance variable in the new Rectangle class.

We can easily define a new class that uses our new trait and is equivalent to Rectangle. Starting with the standard definition of Rectangle, we can edit it to make an equivalent class that uses our new trait:

```
Object subclass: #NewRectangle
  uses: {TRectangle}
  instanceVariableNames: 'origin corner'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'TraitsPaperExample'
```

At this point the browser tells us that NewRectangle still has unsatisfied requirements. Two of the requirements are the methods origin and corner; these methods became necessary when the browser created TRectangle by abstracting direct references to the instance variables of Rectangle and replacing them with message sends. Fortunately, it is easy to provide these methods: all we need do is compile accessors for the corresponding instance variables (see figure 4).

We also need to provide appropriate methods for =, hash and printOn:, the three methods that we decided should not be part of TRectangle. These can be copied from class Rectangle. At this point NewRectangle and Rectangle should be equivalent; after a suitable testing regimen, the class Rectangle could be removed from the system and replaced with NewRectangle.

### 3.3 Conflicting Traits

One of the distinguishing features of traits is that conflicts are *not* automatically resolved when two traits are combined. It is our belief that the complex rules for conflict resolution that accompany most schemes for multiple inheritance are a source of unexpected behavior and a major reason that multiple inheritance is usually avoided. Instead, when a new class (or a new trait) is built from two component traits, any method with different definitions in the two traits results in a trait conflict. A marker method with body self traitConflict is created, and the selector placed in the virtual category *-conflicts-* and coloured red.

It is the programmer's responsibility to resolve 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. In this resolution process, the row of buttons above the code pane is very helpful because it contains a button for each of the available implementations of a method. The color and emphasis of these buttons indicate the status of the corresponding methods (*e.g.*, blue indicates required methods and buttons for excluded methods are semi-transparent).

Once all of the conflicts have been resolved, the meaning of the program should be clear even to a casual observer, who can choose to view the program in a way that hides all of the traits and presents only conventional Squeak classes.

## 4 Implementation

In this section we describe how the *requires* set is computed efficiently enough for it to be displayed in real-time. While computing *requires* is fairly simple for traits, the presence of super-sends makes the computation much more subtle and interesting for classes, so that is where we will focus the discussion. For simplicity, we assume that classes are built in the traditional way, without using traits. Once the basic concept is understood, generalizing it to classes with traits is straightforward.

### 4.1 Computing the Requirements

To compute the required methods of a class we must consider all of the class’s *reachable* methods, that is,

- (1) all the methods that are locally defined in the class,
- (2) all the non-overridden methods defined in its superclasses, and
- (3) all methods that may be reached by super-sends from other reachable methods.

The *requires* set of a class then contains all the selectors subject to a self-send in one of the reachable methods, minus the selectors provided by the class (and by its superclasses). These definitions have been formalized in a technical report [12].

In order to compute this set, we first must find the self-sends and super-sends of a method. Whereas the super-sends can be immediately retrieved from the byte-code, computing the self-sends is more complicated, because they do not all emanate from a single syntactic construct. Consider, for example, the following method:

```
fasten  
  | anObject |  
  self hook.  
  anObject := self.  
  anObject button.  
  self class new clip.
```

From the byte-code of this method, it is immediately clear that `hook` is a self-send. What about `button` and `clip`? These messages are also sent to an instance of the current class, and so they are really self-sends too. However, detecting this requires a deep analysis of this method, as well as of the method `new` on the class side.

Our current implementation does not carry out such an analysis. This means that in the above method, `hook` is the only self-send that we would detect. In order to compensate for this deficiency, we allow the programmer to declare *explicit requirements* for a method.

Even with this simplification, computing the requires set in real-time is quite challenging. The main problem is that a single change in a class may affect the requires set of all its subclasses. Thus, a change in `Object` may mean that we have to update the required methods of all of the classes in the system. A naive implementation based on the above definition would be far too slow to provide the programmer with useful feedback (see section 4.1.4 for a performance comparison). Updating the requires set in real-time required an optimized algorithm that caches critical data and takes advantage of the coherence of the inheritance hierarchy.

#### 4.1.1 *Caching Self-Sends and Super-Sends*

Because computing the self-sends of a single method requires some time-consuming analysis of the code, and because recent Squeak images contain more than 60,000 methods, we decided to cache the self-sends for every method. This means that the self-sends of a method are inferred only once: when the method is first created.

When computing the requires set, looking for methods that contain a certain self-send is far more common than looking for (or modifying) the self-sends of a particular method. Therefore we index the caches by the sent selectors: for each class *C* we maintain a dictionary whose keys are the selectors that are self-sent by the methods directly implemented in *C*, and whose values are arrays of selectors for the methods that perform those self-sends. For example, if the selector `x` is subject to a self-send by the local methods `y` and `z`, looking up `x` returns the array  `#(y z)`.

Super-sends are critical for determining the set of reachable methods, so we also maintain a cache of the super-sends that are issued by the local methods of each class; this cache is similarly indexed by the super-sent selectors rather than by the selectors of the methods that contain the super-sends.

#### 4.1.2 *Using the coherence of the inheritance hierarchy*

When a method is added, modified or removed, we need to check its class, and all its subclasses, to see whether there is any effect on the requirements. The heart of this computation is checking whether a given selector is self-sent in a given class. Especially in the case of large hierarchies, performing this check separately for each subclass proved to be far too slow. Instead, we developed an algorithm that

is recursively applied to all the classes in a hierarchy and takes advantage of the coherence that typically exists between related classes.

*Main method.* The main method of the algorithm recurses through the argument class  $C$  and all its subclasses and finds which of these classes self-sends the selector  $x$ . In addition to  $C$  and  $x$ , this method also takes an argument  $S$ , a set containing the superclass methods that are known to issue self-sends to  $x$ . This set is empty when the method is called on the first class of the hierarchy. The method proceeds as follows:

- (1) We identify  $S' \subseteq S$  that are reachable from the class  $C$ . This means checking which methods in  $S$  are not overridden in  $C$ , or are reachable by a super-send from a method in  $C$ . If  $S'$  is not empty, we directly go to step (3).
- (2) We call the helper method discussed below to search for reachable methods that issue a self-send to the selector  $x$ . The result is stored as  $S'$ .
- (3) If  $S'$  is not empty, we indicate that the class  $C$  self-sends the selector  $x$ . Otherwise, we indicate that  $C$  does not self-send  $x$ .
- (4) We perform a recursive call for each of the direct subclasses of  $C$ , passing  $S'$  as a parameter.

*Helper method.* The purpose of the helper method is to find a set of methods that contain self-sends to a selector  $x$  and are reachable from the class  $C$ . In addition to  $C$  and  $x$ , this method takes a third argument  $U$ , a set of selectors that have been found to be unreachable. This set is empty when the method is called from the main method. The computation works as follows:

- (1) We check whether the class  $C$  contains local methods that issue a self-send to  $x$  and whose selectors are not in the set  $U$  of unreachable selectors. If we find some, we return them and exit.
- (2) Before we use recursion to search the superclass methods, we construct the set  $U'$  of all the superclass selectors that are unreachable from the class for which this helper method was initially called. Since all the unreachable selectors in  $C$  remain unreachable, the set  $U'$  includes all the selectors in  $U$ . In addition,  $U'$  contains all the selectors that are defined in  $C$  (and therefore potentially override superclass methods) and are not super-sent in  $C$ .
- (3) We use recursion to find superclass methods that contain a self-send to  $x$ . In order to avoid finding unreachable methods, we pass the set  $U'$  as a parameter. These methods are stored as  $R$ .
- (4) For each method in the set  $R$ , we check whether it is reachable by a super-send. If so, we replace it by the local method that issued the super-send. Then, we return the resulting set  $R$ .

### 4.1.3 Discussion

The above description conveys the basic idea of our algorithm, but omits several details. In the following, we briefly discuss some of these details and also make some general remarks.

**Cache access.** We see that the caches are well-suited to the algorithm. In step (1) of the main method and steps (2) and (4) of the helper method, we are able to check whether a selector is reachable via a super-send with a single lookup in the super-send dictionary. Similarly, in step (1) of the helper method, we can find all the local methods that issue a self-send to  $x$  by a single lookup in the self-send dictionary.

**Cache consistency.** The caches can be kept in a consistent state quite cheaply. This is mainly because the caches contain only local data, that is, the cache for a particular class is independent of all the other classes in the hierarchy. Thus, modifying a class requires updating of at most the local cache for that class. Furthermore, changing the place of a class in the hierarchy does not affect the caches at all.

**Optimization of the helper method.** In our implementation, we use an optimized version of the helper method described above. The main difference is that we employ temporary caches to avoid doing the computation in step (3) multiple times. Without this optimization, the helper method might, for example, compute the self-sends to a given selector in class Object over and over again, even though it may already be clear that no such self-sends exist. In this context, it is important to note that the helper method does not in general return *all* the reachable methods that contain self-sends to the given selector. However, it is guaranteed to return at least one such method if any exist.

#### 4.1.4 Performance Comparison

Our first implementation of the browser deployed caches for the self-sends and super-sends of every method. However, unlike the approach presented above, these caches were indexed by the selectors of the methods containing the self-sends rather than by the selectors that were subject to the self-sends. Furthermore, our initial algorithm did not take advantage of the coherence in the class hierarchy. This meant that the method for finding out whether a class  $C$  requires a selector  $x$  was applied to each class separately. Using this implementation to find out which classes in the system required the selector  $x$  took several minutes (188 seconds)<sup>1</sup>, which made it impossible to provide immediate feedback.

In a second version of the algorithm, we used the same caching strategy, but took advantage of the coherence in the class hierarchy. This significantly improved the performance of the computation, but it still took over 9 seconds. Finally, using the caching strategy and the algorithms presented above, the same test takes less than 100 milliseconds and therefore meets our requirement for instantaneous feedback.

---

<sup>1</sup> All the performance data provided in this paper were measured in a Squeak 3.2 image consisting of 1860 classes, and were executed on a Pentium III 1.2GHz with 512MB RAM.

## 5 Related and Future Work

The idea of keeping an automatically updated list of things that remain to do dates back at least as far as the “grass catcher” of Trellis [8], and has been adopted in some form or other in many IDEs; the “Tasks” window of Eclipse is another example. However, such lists are typically a by-product of a global re-compilation, rather than being constructed modularly as the consequence of a change to a single method, as in Trellis and in our virtual categories.

Recently, there have been other extensions to Smalltalk browsers that provide the programmer with automatically updated information about the code. In the introduction, we mentioned that there are several Smalltalks that decorate method names in the browser to indicate things such as super-sends or overrides.

Another interesting browser extension is the StarBrowser [16], which is available for VisualWorks and Squeak. The StarBrowser is built on top of a lightweight classification model that allows one to categorize any sort of item (*e.g.*, objects). Besides *extensional classifications* that are just bags of items, the model also allows one to express *intentional classifications*, which are defined by a description, and whose contents are automatically updated.

Despite these improvements, we believe that there remains a large and unexploited field of extensions to the Smalltalk programming environment, which we plan to explore. One example is extending the browser with permanent error indicators that are assigned automatically to a method when an error occurs, and which allow the user to later “replay” the erroneous behavior. In addition, we would like to refine some features that are already available in our current browser, for example, using dataflow analysis to do a better job of inferring self-sends, or extracting traits from smaller browsable units than classes.

## 6 Conclusion

We have identified five virtual categories that, if updated in real-time, provide valuable insights on the program under development. In fact, this categorization gives the programmer a different view on the code: it structures the space of methods in a way that is quite different from the explicitly declared categories (protocols). Whereas the protocols group the methods according to their role in the domain logic (*i.e.*, testing, printing, model access, *etc.*), the virtual categories group the methods according to their role in the composition. In this context, the term “composition” means the way classes are composed from other entities, and includes both inheritance and trait composition.

Our experience with this extended browser has shown that this viewpoint is very valuable both for writing and for understanding the code. While writing, it supports an incremental style of programming: the programmer can freely compose compo-

nents (*i.e.*, inherit from classes and use traits) and add methods while the browser maintains an overview of what still remains to be done and where possible problems (*e.g.*, open requirements, conflicts, and overrides) might lie. Later, the same view helps the programmer to understand the code, because at one glance she can see all the critical methods that are essential for understanding the interaction between the various components. This stands in contrast to the conventional viewpoint, which leaves the programmer the task of finding these critical methods by looking through many methods (sometimes hundreds), spread over many protocols.

From an implementation point of view, most of these categories are quite easy to compute. By far the most challenging is the computation of the requires set. There are three main reasons for this. First, the absence of explicit type information makes it hard to detect the requirements; second, finding whether a method is reachable is not so trivial as it might seem; and third, heavy optimization is required to attain real-time performance.

Nevertheless, our experience has shown that these obstacles can be overcome, and have only a minimal impact on the usefulness of the extended browser. Although we choose a very simple algorithm to infer self-sends, in practice most of the requirements are detected. Furthermore, caching of information about the requirements means that the requires set can be quickly re-computed even after a change that affects many classes.

## References

- [1] Andrew Black, Nathanael Schärli, and Stéphane Ducasse. Applying traits to the Smalltalk collection hierarchy. Technical Report IAM-02-007, Institut für Informatik, Universität Bern, Switzerland, November 2002. Also available as Technical Report CSE-02-014, OGI School of Science & Engineering, Beaverton, Oregon, USA.
- [2] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices*, pages 303–311, October 1990. Published as *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices*, volume 25, number 10.
- [3] Eclipse Platform: Technical Overview, 2003. <http://www.eclipse.org/whitepapers/eclipse-overview.pdf>.
- [4] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA'97*, pages 318–326, November 1997.
- [5] Ron Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed*. Addison Wesley, 2001.
- [6] Niraj Jetly. VisualAge for Java 2.0. *Java Developer's Journal*, 4(4):48–49, April 1999.
- [7] Edward J. Klimas. Getting the biggest bang for your buck. *Visual Age Magazine*, May 1998.

- [8] Patrick D. O'Brien, Daniel C. Halbert, and Michael F. Kilian. The Trellis programming environment. In *Proceedings Object-oriented programming systems, languages and applications, ACM SIGPLAN Notices*, volume 22, pages 91–102. ACM Press, October 1987.
- [9] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.
- [10] Nathanael Schärli. Traits prototype in Squeak, February 2003. <http://www.iam.unibe.ch/~schaerli/smalltalk/traits/traitsPrototype.htm>.
- [11] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003*, LNCS. Springer Verlag, July 2003. to appear.
- [12] Nathanael Schärli, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, and Andrew Black. Traits: The formal model. Technical Report IAM-02-006, Institut für Informatik, Universität Bern, Switzerland, November 2002. Also available as Technical Report CSE-02-013, OGI School of Science & Engineering, Beaverton, Oregon, USA.
- [13] Squeak. <http://www.squeak.org/>.
- [14] L. Tesler. The Smalltalk environment. *Byte*, 6(8), August 1981.
- [15] Cincom VisualWorks Smalltalk. <http://www.cincom.com/scripts/smalltalk.dll/>.
- [16] Roel Wuyts. StarBrowser. <http://www.iam.unibe.ch/~wuyts/StarBrowser/>.